`io` — Core tools for working with streams

Source code: Lib/io.py

Overview

The `io` module provides Python's main facilities for dealing with various types of I/O. There are three main types of I/O: *text I/O*, *binary I/O* and *raw I/O*. These are generic categories, and various backing stores can be used for each of them. A concrete object belonging to any of these categories is called a file object. Other common terms are *stream* and *file-like object*.

Independent of its category, each concrete stream object will also have various capabilities: it can be read-only, write-only, or read-write. It can also allow arbitrary random access (seeking forwards or backwards to any location), or only sequential access (for example in the case of a socket or pipe).

All streams are careful about the type of data you give to them. For example giving a `str`object to the `write()` method of a binary stream will raise a `TypeError`. So will giving a `bytes` object to the `write()` method of a text stream.

Changed in version 3.3: Operations that used to raise `IOError` now raise `OSError`, since`IOError` is now an alias of `OSError`.

Text I/O

Text I/O expects and produces `str` objects. This means that whenever the backing store is natively made of bytes (such as in the case of a file), encoding and decoding of data is made transparently as well as optional translation of platform-specific newline characters.

The easiest way to create a text stream is with `open()`, optionally specifying an encoding:

```
f = open("myfile.txt", "r", encoding="utf-8")
```

In-memory text streams are also available as `StringIO` objects:

```
f = io.StringIO("some initial text data")
```

The text stream API is described in detail in the documentation of `TextIOBase`.

Binary I/O

Binary I/O (also called *buffered I/O*) expects bytes-like objects and produces `bytes` objects. No encoding, decoding, or newline translation is performed. This category of streams can be used for all kinds of non-text data, and also when manual control over the handling of text data is desired.

The easiest way to create a binary stream is with `open()` with `'b'` in the mode string:

```
f = open("myfile.jpg", "rb")
```

In-memory binary streams are also available as `BytesIO` objects:

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

The binary stream API is described in detail in the docs of `BufferedIOBase`.

Other library modules may provide additional ways to create text or binary streams. See `socket.socket.makefile()` for example.

Raw I/O

Raw I/O (also called *unbuffered I/O*) is generally used as a low-level building-block for binary and text streams; it is rarely useful to directly manipulate a raw stream from user code. Nevertheless, you can create a raw stream by opening a file in binary mode with buffering disabled:

```
f = open("myfile.jpg", "rb", buffering=0)
```

The raw stream API is described in detail in the docs of `RawIOBase`.

High-level Module Interface

`io.`**`DEFAULT_BUFFER_SIZE`**
An int containing the default buffer size used by the module's buffered I/O classes.`open()` uses the file's blksize (as obtained by `os.stat()`) if possible.

`io.`**`open`**(*`file`*, *`mode='r'`*, *`buffering=-1`*, *`encoding=None`*, *`errors=None`*, *`newline=None`*, *`closefd=True`*, *`opener=None`*)
This is an alias for the builtin `open()` function.

This function raises an auditing event `open` with arguments `path`, `mode` and `flags`. The `mode` and `flags` arguments may have been modified or inferred from the original call.

`io.`**`open_code`**(*`path`*)

Opens the provided file with mode `'rb'`. This function should be used when the intent is to treat the contents as executable code.

`path` should be a `str` and an absolute path.

The behavior of this function may be overridden by an earlier call to the`PyFile_SetOpenCodeHook()`. However, assuming that `path` is a `str` and an absolute path, `open_code(path)` should always behave the same as `open(path, 'rb')`. Overriding the behavior is intended for additional validation or preprocessing of the file.

New in version 3.8.

exception `io.`**`BlockingIOError`**

This is a compatibility alias for the builtin `BlockingIOError` exception.

exception `io.`**`UnsupportedOperation`**

An exception inheriting `OSError` and `ValueError` that is raised when an unsupported operation is called on a stream.

In-memory streams

It is also possible to use a `str` or bytes-like object as a file for both reading and writing. For strings `StringIO` can be used like a file opened in text mode. `BytesIO` can be used like a file opened in binary mode. Both provide full read-write capabilities with random access.

See also

`sys`
contains the standard IO streams: `sys.stdin`, `sys.stdout`, and `sys.stderr`.

Class hierarchy

The implementation of I/O streams is organized as a hierarchy of classes. First abstract base classes (ABCs), which are used to specify the various categories of streams, then concrete classes providing the standard stream implementations.

> **Note** The abstract base classes also provide default implementations of some methods in order to help implementation of concrete stream classes. For example, `BufferedIOBase` provides unoptimized implementations of`readinto()` and `readline()`.

At the top of the I/O hierarchy is the abstract base class `IOBase`. It defines the basic interface to a stream. Note, however, that there is no separation between reading and writing to streams; implementations are allowed to raise `UnsupportedOperation` if they do not support a given operation.

The `RawIOBase` ABC extends `IOBase`. It deals with the reading and writing of bytes to a stream. `FileIO` subclasses `RawIOBase` to provide an interface to files in the machine's file system.

The `BufferedIOBase` ABC deals with buffering on a raw byte stream (`RawIOBase`). Its subclasses, `BufferedWriter`, `BufferedReader`, and `BufferedRWPair` buffer streams that are readable, writable, and both readable and writable. `BufferedRandom` provides a buffered interface to random access streams. Another `BufferedIOBase` subclass, `BytesIO`, is a stream of in-memory bytes.

The `TextIOBase` ABC, another subclass of `IOBase`, deals with streams whose bytes represent text, and handles encoding and decoding to and from strings. `TextIOWrapper`, which extends it, is a buffered text interface to a buffered raw stream (`BufferedIOBase`). Finally, `StringIO` is an in-memory stream for text.

Argument names are not part of the specification, and only the arguments of `open()` are intended to be used as keyword arguments.

The following table summarizes the ABCs provided by the `io` module:

ABC	Inherits	Stub Methods	Mixin Methods and Properties
`IOBase`		`fileno`, `seek`, and `truncate`	`close`, `closed`, `__enter__`,`__exit__`, `flush`, `isatty`, `__iter__`, `__next__`, `readable`, `rea` and `writelines`
`RawIOBase`	`IOBase`	`readinto` and`write`	Inherited `IOBase` methods, `read`, and `readall`
`BufferedIOBase`	`IOBase`	`detach`, `read`,`read1`, and `write`	Inherited `IOBase` methods, `readinto`, and `readinto1`
`TextIOBase`	`IOBase`	`detach`, `read`,`readline`, and`write`	Inherited `IOBase` methods, `encoding`, `errors`, and `newlines`

I/O Base Classes

class `io.`**`IOBase`**

The abstract base class for all I/O classes, acting on streams of bytes. There is no public constructor.

This class provides empty abstract implementations for many methods that derived classes can override selectively; the default implementations represent a file that cannot be read, written or seeked.

Even though `IOBase` does not declare `read()` or `write()` because their signatures will vary, implementations and clients should consider those methods part of the interface. Also, implementations may raise a `ValueError` (or `UnsupportedOperation`) when operations they do not support are called.

The basic type used for binary data read from or written to a file is `bytes`. Other bytes-like objects are accepted as method arguments too. Text I/O classes work with `str` data.

Note that calling any method (even inquiries) on a closed stream is undefined. Implementations may raise `ValueError` in this case.

`IOBase` (and its subclasses) supports the iterator protocol, meaning that an `IOBase` object can be iterated over yielding the lines in a stream. Lines are defined slightly differently depending on whether the stream is a binary stream (yielding bytes), or a text stream (yielding character strings). See `readline()` below.

`IOBase` is also a context manager and therefore supports the `with` statement. In this example, *file* is closed after the `with` statement's suite is finished—even if an exception occurs:

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

`IOBase` provides these data attributes and methods:

close()

Flush and close this stream. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise a `ValueError`.

As a convenience, it is allowed to call this method more than once; only the first call, however, will have an effect.

closed

`True` if the stream is closed.

fileno()

Return the underlying file descriptor (an integer) of the stream if it exists. An `OSError`is raised if the IO object does not use a file descriptor.

flush()

Flush the write buffers of the stream if applicable. This does nothing for read-only and non-blocking streams.

isatty()

Return `True` if the stream is interactive (i.e., connected to a terminal/tty device).

readable()

Return `True` if the stream can be read from. If `False`, `read()` will raise `OSError`.

readline(*size=-1*)

Read and return one line from the stream. If *size* is specified, at most *size* bytes will be read.

The line terminator is always `b'\n'` for binary files; for text files, the *newline*argument to `open()` can be used to select the line terminator(s) recognized.

readlines(*hint=-1*)

Read and return a list of lines from the stream. *hint* can be specified to control the number of lines read: no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds *hint*.

Note that it's already possible to iterate on file objects using `for line in file: ...`without calling `file.readlines()`.

seek(*offset*, *whence=SEEK_SET*)

Change the stream position to the given byte *offset*. *offset* is interpreted relative to the position indicated by *whence*. The default value for *whence* is `SEEK_SET`. Values for *whence* are:

- `SEEK_SET` or `0` – start of the stream (the default); *offset* should be zero or positive
- `SEEK_CUR` or `1` – current stream position; *offset* may be negative
- `SEEK_END` or `2` – end of the stream; *offset* is usually negative

Return the new absolute position.

New in version 3.1: The `SEEK_*` constants.

New in version 3.3: Some operating systems could support additional values, like`os.SEEK_HOLE` or `os.SEEK_DATA`. The valid values for a file could depend on it being open in text or binary mode.

seekable()

Return `True` if the stream supports random access. If `False`, `seek()`, `tell()` and `truncate()` will raise `OSError`.

tell()

Return the current stream position.

truncate(*size=None*)
Resize the stream to the given *size* in bytes (or the current position if *size* is not specified). The current stream position isn't changed. This resizing can extend or reduce the current file size. In case of extension, the contents of the new file area depend on the platform (on most systems, additional bytes are zero-filled). The new file size is returned.

Changed in version 3.5: Windows will now zero-fill files when extending.

writable()
Return `True` if the stream supports writing. If `False`, `write()` and `truncate()` will raise `OSError`.

writelines(*lines*)
Write a list of lines to the stream. Line separators are not added, so it is usual for each of the lines provided to have a line separator at the end.

__del__()
Prepare for object destruction. `IOBase` provides a default implementation of this method that calls the instance's `close()` method.

class `io.`**RawIOBase**
Base class for raw binary I/O. It inherits `IOBase`. There is no public constructor.

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 left to Buffered I/O and Text I/O, described later in this page).

In addition to the attributes and methods from `IOBase`, `RawIOBase` provides the following methods:

read(*size=-1*)
Read up to *size* bytes from the object and return them. As a convenience, if *size* is unspecified or -1, all bytes until EOF are returned. Otherwise, only one system call is ever made. Fewer than *size* bytes may be returned if the operating system call returns fewer than *size* bytes.

If 0 bytes are returned, and *size* was not 0, this indicates end of file. If the object is in non-blocking mode and no bytes are available, `None` is returned.

The default implementation defers to `readall()` and `readinto()`.

readall()
Read and return all the bytes from the stream until EOF, using multiple calls to the stream if necessary.

readinto(*b*)
Read bytes into a pre-allocated, writable bytes-like object *b*, and return the number of bytes read. For example, *b* might be a `bytearray`. If the object is in non-blocking mode and no bytes are available, `None` is returned.

write(*b*)
Write the given bytes-like object, *b*, to the underlying raw stream, and return the number of bytes written. This can be less than the length of *b* in bytes, depending on specifics of the underlying raw stream, and especially if it is in non-blocking mode.`None` is returned if the raw stream is set not to block and no single byte could be readily written to it. The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

class `io.`**BufferedIOBase**
Base class for binary streams that support some kind of buffering. It inherits `IOBase`. There is no public constructor.

The main difference with `RawIOBase` is that methods `read()`, `readinto()` and `write()`will try (respectively) to read as much input as requested or to consume all given output, at the expense of making perhaps more than one system call.

In addition, those methods can raise `BlockingIOError` if the underlying raw stream is in non-blocking mode and cannot take or give enough data; unlike their `RawIOBase`counterparts, they will never return `None`.

Besides, the `read()` method does not have a default implementation that defers to `readinto()`.

A typical `BufferedIOBase` implementation should not inherit from a `RawIOBase`implementation, but wrap one, like `BufferedWriter` and `BufferedReader` do.

`BufferedIOBase` provides or overrides these methods and attribute in addition to those from `IOBase`:

raw
The underlying raw stream (a `RawIOBase` instance) that `BufferedIOBase` deals with. This is not part of the `BufferedIOBase` API and may not exist on some implementations.

detach()
Separate the underlying raw stream from the buffer and return it.

After the raw stream has been detached, the buffer is in an unusable state.

Some buffers, like `BytesIO`, do not have the concept of a single raw stream to return from this method. They raise `UnsupportedOperation`.

New in version 3.1.

read(*size=-1*)

Read and return up to *size* bytes. If the argument is omitted, `None`, or negative, data is read and returned until EOF is reached. An empty `bytes` object is returned if the stream is already at EOF.

If the argument is positive, and the underlying raw stream is not interactive, multiple raw reads may be issued to satisfy the byte count (unless EOF is reached first). But for interactive raw streams, at most one raw read will be issued, and a short result does not imply that EOF is imminent.

A `BlockingIOError` is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

read1([*size*])

Read and return up to *size* bytes, with at most one call to the underlying raw stream's `read()` (or `readinto()`) method. This can be useful if you are implementing your own buffering on top of a `BufferedIOBase` object.

If *size* is `-1` (the default), an arbitrary number of bytes are returned (more than zero unless EOF is reached).

readinto(*b*)

Read bytes into a pre-allocated, writable bytes-like object *b* and return the number of bytes read. For example, *b* might be a `bytearray`.

Like `read()`, multiple reads may be issued to the underlying raw stream, unless the latter is interactive.

A `BlockingIOError` is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

readinto1(*b*)

Read bytes into a pre-allocated, writable bytes-like object *b*, using at most one call to the underlying raw stream's `read()` (or `readinto()`) method. Return the number of bytes read.

A `BlockingIOError` is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

New in version 3.5.

write(*b*)

Write the given bytes-like object, *b*, and return the number of bytes written (always equal to the length of *b* in bytes, since if the write fails an `OSError` will be raised). Depending on the actual implementation, these bytes may be readily written to the underlying stream, or held in a buffer for performance and latency reasons.

When in non-blocking mode, a `BlockingIOError` is raised if the data needed to be written to the raw stream but it couldn't accept all the data without blocking.

The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

Raw File I/O

class `io.`**FileIO**(*name*, *mode='r'*, *closefd=True*, *opener=None*)

`FileIO` represents an OS-level file containing bytes data. It implements the `RawIOBase`interface (and therefore the `IOBase` interface, too).

The *name* can be one of two things:

- a character string or `bytes` object representing the path to the file which will be opened. In this case closefd must be `True` (the default) otherwise an error will be raised.
- an integer representing the number of an existing OS-level file descriptor to which the resulting `FileIO` object will give access. When the FileIO object is closed this fd will be closed as well, unless *closefd* is set to `False`.

The *mode* can be `'r'`, `'w'`, `'x'` or `'a'` for reading (default), writing, exclusive creation or appending. The file will be created if it doesn't exist when opened for writing or appending; it will be truncated when opened for writing. `FileExistsError` will be raised if it already exists when opened for creating. Opening a file for creating implies writing, so this mode behaves in a similar way to `'w'`. Add a `'+'` to the mode to allow simultaneous reading and writing.

The `read()` (when called with a positive argument), `readinto()` and `write()` methods on this class will only make one system call.

A custom opener can be used by passing a callable as *opener*. The underlying file descriptor for the file object is then obtained by calling *opener* with (*name*, *flags*). *opener*must return an open file descriptor (passing `os.open` as *opener* results in functionality similar to passing `None`).

The newly created file is non-inheritable.

See the `open()` built-in function for examples on using the *opener* parameter.

Changed in version 3.3: The *opener* parameter was added. The `'x'` mode was added.

Changed in version 3.4: The file is now non-inheritable.

In addition to the attributes and methods from `IOBase` and `RawIOBase`, `FileIO` provides the following data attributes:

mode
The mode as given in the constructor.

name
The file name. This is the file descriptor of the file when no name is given in the constructor.

Buffered Streams

Buffered I/O streams provide a higher-level interface to an I/O device than raw I/O does.

class `io.`**BytesIO**([*initial_bytes*])
A stream implementation using an in-memory bytes buffer. It inherits `BufferedIOBase`. The buffer is discarded when the `close()` method is called.

The optional argument *initial_bytes* is a bytes-like object that contains initial data.

`BytesIO` provides or overrides these methods in addition to those from `BufferedIOBase`and `IOBase`:

getbuffer()
Return a readable and writable view over the contents of the buffer without copying them. Also, mutating the view will transparently update the contents of the buffer:

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

Note As long as the view exists, the `BytesIO` object cannot be resized or closed.

New in version 3.2.

getvalue()
Return `bytes` containing the entire contents of the buffer.

read1([*size*])
In `BytesIO`, this is the same as `read()`.

Changed in version 3.7: The *size* argument is now optional.

readinto1(*b*)
In `BytesIO`, this is the same as `readinto()`.

New in version 3.5.

class `io.`**BufferedReader**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
A buffer providing higher-level access to a readable, sequential `RawIOBase` object. It inherits `BufferedIOBase`. 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.

The constructor creates a `BufferedReader` for the given readable *raw* stream and *buffer_size*. If *buffer_size* is omitted, `DEFAULT_BUFFER_SIZE` is used.

`BufferedReader` provides or overrides these methods in addition to those from `BufferedIOBase` and `IOBase`:

peek([*size*])
Return bytes from the stream without advancing the position. At most one single read on the raw stream is done to satisfy the call. The number of bytes returned may be less or more than requested.

read([*size*])
Read and return *size* bytes, or if *size* is not given or negative, until EOF or if the read call would block in non-blocking mode.

read1([*size*])
Read and return up to *size* bytes with only one call on the raw stream. If at least one byte is buffered, only buffered bytes are returned. Otherwise, one raw stream read call is made.

Changed in version 3.7: The *size* argument is now optional.

class `io.`**BufferedWriter**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
A buffer providing higher-level access to a writeable, sequential `RawIOBase` object. It inherits `BufferedIOBase`. 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 a `seek()` is requested (for `BufferedRandom` objects);
- when the `BufferedWriter` object is closed or destroyed.

The constructor creates a `BufferedWriter` for the given writeable *raw* stream. If the *buffer_size* is not given, it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedWriter` provides or overrides these methods in addition to those from `BufferedIOBase` and `IOBase`:

`flush`()
Force bytes held in the buffer into the raw stream. A `BlockingIOError` should be raised if the raw stream blocks.

`write`(*b*)
Write the bytes-like object, *b*, and return the number of bytes written. When in non-blocking mode, a `BlockingIOError` is raised if the buffer needs to be written out but the raw stream blocks.

class `io.`**`BufferedRandom`**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
A buffered interface to random access streams. It inherits `BufferedReader` and `BufferedWriter`.

The constructor creates a reader and writer for a seekable raw stream, given in the first argument. If the *buffer_size* is omitted it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedRandom` is capable of anything `BufferedReader` or `BufferedWriter` can do. In addition, `seek()` and `tell()` are guaranteed to be implemented.

class `io.`**`BufferedRWPair`**(*reader*, *writer*, *buffer_size=DEFAULT_BUFFER_SIZE*)
A buffered I/O object combining two unidirectional `RawIOBase` objects – one readable, the other writeable – into a single bidirectional endpoint. It inherits `BufferedIOBase`.

reader and *writer* are `RawIOBase` objects that are readable and writeable respectively. If the *buffer_size* is omitted it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedRWPair` implements all of `BufferedIOBase`'s methods except for `detach()`, which raises `UnsupportedOperation`.

> **Warning** `BufferedRWPair` does not attempt to synchronize accesses to its underlying raw streams. You should not pass it the same object as reader and writer; use `BufferedRandom` instead.

Text I/O

class `io.`**`TextIOBase`**
Base class for text streams. This class provides a character and line based interface to stream I/O. It inherits `IOBase`. There is no public constructor.

`TextIOBase` provides or overrides these data attributes and methods in addition to those from `IOBase`:

`encoding`
The name of the encoding used to decode the stream's bytes into strings, and to encode strings into bytes.

`errors`
The error setting of the decoder or encoder.

`newlines`
A string, a tuple of strings, or `None`, indicating the newlines translated so far. Depending on the implementation and the initial constructor flags, this may not be available.

`buffer`
The underlying binary buffer (a `BufferedIOBase` instance) that `TextIOBase` deals with. This is not part of the `TextIOBase` API and may not exist in some implementations.

`detach`()
Separate the underlying binary buffer from the `TextIOBase` and return it.

After the underlying buffer has been detached, the `TextIOBase` is in an unusable state.

Some `TextIOBase` implementations, like `StringIO`, may not have the concept of an underlying buffer and calling this method will raise `UnsupportedOperation`.

New in version 3.1.

`read`(*size=-1*)
Read and return at most *size* characters from the stream as a single `str`. If *size* is negative or `None`, reads until EOF.

`readline`(*size=-1*)
Read until newline or EOF and return a single `str`. 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=SEEK_SET*)
Change the stream position to the given *offset*. Behaviour 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).

Return the new absolute position as an opaque number.

New in version 3.1: The SEEK_* constants.

tell()
Return the current stream position as an opaque number. The number does not usually represent a number of bytes in the underlying binary storage.

write(*s*)
Write the string *s* to the stream and return the number of characters written.

class io.**TextIOWrapper**(*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, *write_through=False*)
A buffered text stream over a BufferedIOBase binary stream. It inherits TextIOBase.

encoding gives the name of the encoding that the stream will be decoded or encoded with. It defaults to locale.getpreferredencoding(False).

errors is an optional string that specifies how encoding and decoding errors are to be handled. Pass 'strict' to raise a ValueError exception if there is an encoding error (the default of None has the same effect), or pass 'ignore' to ignore errors. (Note that ignoring encoding errors can lead to data loss.) 'replace' causes a replacement marker (such as '?') to be inserted where there is malformed data. 'backslashreplace'causes malformed data to be replaced by a backslashed escape sequence. When writing, 'xmlcharrefreplace' (replace with the appropriate XML character reference) or 'namereplace' (replace with \N{...} escape sequences) can be used. Any other error handling name that has been registered with codecs.register_error() is also valid.

newline controls how line endings are handled. It can be None, '', '\n', '\r', and '\r\n'. It works as follows:

- When reading input from the stream, if *newline* is None, universal newlines mode is enabled. Lines in the input can end in '\n', '\r', or '\r\n', and these are translated into '\n' before being returned to the caller. If it is '', universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- When writing output to the stream, if *newline* is None, any '\n' characters written are translated to the system default line separator, os.linesep. If *newline* is '' or '\n', no translation takes place. If *newline* is any of the other legal values, any '\n' characters written are translated to the given string.

If *line_buffering* is True, flush() is implied when a call to write contains a newline character or a carriage return.

If *write_through* is True, calls to write() are guaranteed not to be buffered: any data written on the TextIOWrapper object is immediately handled to its underlying binary *buffer*.

Changed in version 3.3: The *write_through* argument has been added.

Changed in version 3.3: The default *encoding* is now locale.getpreferredencoding(False) instead of locale.getpreferredencoding(). Don't change temporary the locale encoding using locale.setlocale(), use the current locale encoding instead of the user preferred encoding.

TextIOWrapper provides these members in addition to those of TextIOBase and its parents:

line_buffering
Whether line buffering is enabled.

write_through
Whether writes are passed immediately to the underlying binary buffer.

New in version 3.7.

reconfigure(**, encoding][, errors][, newline][, line_buffering][, write_through]*)
Reconfigure this text stream using new settings for *encoding*, *errors*, *newline*, *line_buffering* and *write_through*.

Parameters not specified keep current settings, except errors='strict' is used when *encoding* is specified but *errors* is not specified.

It is not possible to change the encoding or newline if some data has already been read from the stream. On the other hand, changing encoding after write is possible.

This method does an implicit stream flush before setting the new parameters.

New in version 3.7.

class io.**StringIO**(*initial_value=''*, *newline='\n'*)

An in-memory stream for text I/O. The text buffer is discarded when the `close()` method is called.

The initial value of the buffer can be set by providing *initial_value*. If newline translation is enabled, newlines will be encoded as if by `write()`. The stream is positioned at the start of the buffer.

The *newline* argument works like that of `TextIOWrapper`. The default is to consider only `\n` characters as ends of lines and to do no newline translation. If *newline* is set to `None`, newlines are written as `\n` on all platforms, but universal newline decoding is still performed when reading.

`StringIO` provides this method in addition to those from `TextIOBase` and its parents:

`getvalue`()

Return a `str` containing the entire contents of the buffer. Newlines are decoded as if by `read()`, although the stream position is not changed.

Example usage:

```
import io

output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

class `io.`**`IncrementalNewlineDecoder`**

A helper codec that decodes newlines for universal newlines mode. It inherits `codecs.IncrementalDecoder`.

Performance

This section discusses the performance of the provided concrete I/O implementations.

Binary I/O

By reading and writing only large chunks of data even when the user asks for a single byte, buffered I/O hides any inefficiency in calling and executing the operating system's unbuffered I/O routines. The gain depends on the OS and the kind of I/O which is performed. For example, on some modern OSes such as Linux, unbuffered disk I/O can be as fast as buffered I/O. The bottom line, however, is that buffered I/O offers predictable performance regardless of the platform and the backing device. Therefore, it is almost always preferable to use buffered I/O rather than unbuffered I/O for binary data.

Text I/O

Text I/O over a binary storage (such as a file) is significantly slower than binary I/O over the same storage, because it requires conversions between unicode and binary data using a character codec. This can become noticeable handling huge amounts of text data like large log files. Also, `TextIOWrapper.tell()` and `TextIOWrapper.seek()` are both quite slow due to the reconstruction algorithm used.

`StringIO`, however, is a native in-memory unicode container and will exhibit similar speed to `BytesIO`.

Multi-threading

`FileIO` objects are thread-safe to the extent that the operating system calls (such as `read(2)` under Unix) they wrap are thread-safe too.

Binary buffered objects (instances of `BufferedReader`, `BufferedWriter`, `BufferedRandom` and `BufferedRWPair`) protect their internal structures using a lock; it is therefore safe to call them from multiple threads at once.

`TextIOWrapper` objects are not thread-safe.

Reentrancy

Binary buffered objects (instances of `BufferedReader`, `BufferedWriter`, `BufferedRandom` and `BufferedRWPair`) are not reentrant. While reentrant calls will not happen in normal situations, they can arise from doing I/O in a `signal` handler. If a thread tries to re-enter a buffered object which it is already accessing, a `RuntimeError` is raised. Note this doesn't prohibit a different thread from entering the buffered object.

The above implicitly extends to text files, since the `open()` function will wrap a buffered object inside a `TextIOWrapper`. This includes standard streams and therefore affects the built-in function `print()` as well.