16.2. `io` — Core tools for working with streams¶

New in version 2.6.

The `io` module provides the Python interfaces to stream handling. The built-in `open()` function is defined in this module.

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 throw an `IOError` if they do not support a given operation.

Extending `IOBase` is `RawIOBase` which deals simply with the reading and writing of raw bytes to a stream. `FileIO` subclasses `RawIOBase` to provide an interface to files in the machine's file system.

`BufferedIOBase` 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. `BytesIO` is a simple stream of in-memory bytes.

Another `IOBase` subclass, `TextIOBase`, deals with streams whose bytes represent text, and handles encoding and decoding from and to 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.

16.2.1. 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*[, *buffering*[, *encoding*[, *errors*[, *newline*[, *closefd=True*]]]]]])¶

Open *file* and return a stream. If the file cannot be opened, an `IOError` is raised.

file is either a string giving the name (and the path if the file isn't in the current working directory) of the file to be opened or a file descriptor of the file to be opened. (If a file descriptor is given, for example, from `os.fdopen()`, it is closed when the returned I/O object is closed, unless *closefd* is set to `False`.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode. Other common values are `'w'` for writing (truncating the file if it already exists), and `'a'` for appending (which on *some* Unix systems, means that *all* writes append to the end of the file regardless of the current seek position). In text mode, if *encoding* is not specified the encoding used is platform dependent. (For reading and writing raw bytes use binary mode and leave *encoding* unspecified.) The available modes are:

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'a'`	open for writing, appending to the end of the file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open a disk file for updating (reading and writing)
`'U'`	universal newline mode (for backwards compatibility; should not be used in new code)

The default mode is `'rt'` (open for reading text). For binary random access, the mode `'w+b'` opens and truncates the file to 0 bytes, while `'r+b'` opens the file without truncation.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn't. Files opened in binary mode (including `'b'` in the *mode* argument) return contents as `bytes` objects without any decoding. In text mode (the default, or when `'t'` is included in the *mode* argument), the contents of the file are returned as strings, the bytes having been first decoded using a platform-dependent encoding or using the specified *encoding* if given.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no *buffering* argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on `DEFAULT_BUFFER_SIZE`. On many systems, the buffer will typically be 4096 or 8192 bytes long.
- "Interactive" text files (files for which `isatty()` returns True) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent, but any encoding supported by Python can be used. See the `codecs` module for the list of supported encodings.

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. When writing, `'xmlcharrefreplace'` (replace with the appropriate XML character reference) or `'backslashreplace'` (replace with backslashed escape sequences) can be used. Any other error handling name that has been registered with `codecs.register_error()` is also valid.

newline controls how universal newlines works (it only applies to text mode). It can be `None`, `''`, `'\n'`, `'\r'`, and `'\r\n'`. It works as follows:

- On input, 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 newline 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.
- On output, if *newline* is `None`, any `'\n'` characters written are translated to the system default line separator, `os.linesep`. If *newline* is `''`, no translation takes place. If *newline* is any of the other legal values, any `'\n'` characters written are translated to the given string.

If *closefd* is `False` and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given *closefd* has no effect but must be `True` (the default).

The type of file object returned by the `open()` function depends on the mode. When `open()` is used to open a file in a text mode (`'w'`, `'r'`, `'wt'`, `'rt'`, etc.), it returns a `TextIOWrapper`. When used to open a file in a binary mode, the returned class varies: in read binary mode, it returns a `BufferedReader`; in write binary and append binary modes, it returns a `BufferedWriter`, and in read/write mode, it returns a `BufferedRandom`.

It is also possible to use a string or bytearray as a file for both reading and writing. For strings `StringIO` can be used like a file opened in a text mode, and for bytearrays a `BytesIO` can be used like a file opened in a binary mode.

exception `io.BlockingIOError`¶

Error raised when blocking would occur on a non-blocking stream. It inherits `IOError`.

In addition to those of `IOError`, `BlockingIOError` has one attribute:

`characters_written`¶

An integer containing the number of characters written to the stream before it blocked.

exception `io.UnsupportedOperation`¶

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

16.2.2. 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()`, `readinto()`, or `write()` because their signatures will vary, implementations and clients should consider those methods part of the interface. Also, implementations may raise a `IOError` when operations they do not support are called.

The basic type used for binary data read from or written to a file is `bytes`. `bytearrays` are accepted too, and in some cases (such as `readinto`) required. Text I/O classes work with `str` data.

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

IOBase (and its subclasses) support the iterator protocol, meaning that an `IOBase` object can be iterated over yielding the lines in a stream.

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 an `IOError`. The internal file descriptor isn't closed if *closefd* was False.

`closed`¶

True if the stream is closed.

`fileno`()¶

Return the underlying file descriptor (an integer) of the stream if it exists. An `IOError` 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 `IOError`.

`readline`([*limit*])¶

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

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

`readlines`([*hint*])¶

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*.

`seek`(*offset*[, *whence*])¶

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

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

Return the new absolute position.

`seekable`()¶

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

`tell`()¶

Return the current stream position.

`truncate`([*size*])¶

Truncate the file to at most *size* bytes. *size* defaults to the current file position, as returned by `tell()`.

`writable`()¶

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

`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.

class `io.RawIOBase`¶

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

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

`read`([*n*])¶

Read and return all the bytes from the stream until EOF, or if *n* is specified, up to *n* bytes. Only one system call is ever made. An empty bytes object is returned on EOF; `None` is returned if the object is set not to block and has no data to read.

`readall`()¶

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

`readinto`(*b*)¶

Read up to len(b) bytes into bytearray *b* and return the number of bytes read.

`write`(*b*)¶

Write the given bytes or bytearray object, *b*, to the underlying raw stream and return the number of bytes written (This is never less than `len(b)`, since if the write fails, an `IOError` will be raised).

class `io.BufferedIOBase`¶

Base class for streams that support buffering. It inherits `IOBase`. There is no public constructor.

The main difference with `RawIOBase` is that the `read()` method supports omitting the *size* argument, and does not have a default implementation that defers to `readinto()`.

In addition, `read()`, `readinto()`, and `write()` may raise `BlockingIOError` if the underlying raw stream is in non-blocking mode and not ready; unlike their raw counterparts, they will never return `None`.

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

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

`read`([*n*])¶

Read and return up to *n* 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 has no data at the moment.

`readinto`(*b*)¶

Read up to len(b) bytes into bytearray *b* and return the number of bytes read.

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 has no data at the moment.

`write`(*b*)¶

Write the given bytes or bytearray object, *b*, to the underlying raw stream and return the number of bytes written (never less than `len(b)`, since if the write fails an `IOError` will be raised).

A `BlockingIOError` is raised if the buffer is full, and the underlying raw stream cannot accept more data at the moment.

16.2.3. Raw File I/O¶

class `io.FileIO`(*name*[, *mode*])¶

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

The *mode* can be `'r'`, `'w'` or `'a'` for reading (default), writing, 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. Add a `'+'` to the mode to allow simultaneous reading and writing.

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

`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.

`read`([*n*])¶
Read and return at most *n* bytes. Only one system call is made, so it is possible that less data than was requested is returned. Use `len()` on the returned bytes object to see how many bytes were actually returned. (In non-blocking mode, `None` is returned when no data is available.)

`readall`()¶
Read and return the entire file's contents in a single bytes object. As much as immediately available is returned in non-blocking mode. If the EOF has been reached, `b''` is returned.

`write`(*b*)¶
Write the bytes or bytearray object, *b*, to the file, and return the number actually written. Only one system call is made, so it is possible that only some of the data is written.

Note that the inherited `readinto()` method should not be used on `FileIO` objects.

16.2.4. Buffered Streams¶

class `io.BytesIO`([*initial_bytes*])¶

A stream implementation using an in-memory bytes buffer. It inherits `BufferedIOBase`.

The argument *initial_bytes* is an optional initial bytearray.

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

`getvalue`()¶

Return `bytes` containing the entire contents of the buffer.

`read1`()¶
In `BytesIO`, this is the same as `read()`.

`truncate`([*size*])¶
Truncate the buffer to at most *size* bytes. *size* defaults to the current stream position, as returned by `tell()`.

class `io.BufferedReader`(*raw*[, *buffer_size*])¶

A buffer for a readable, sequential `RawIOBase` object. It inherits `BufferedIOBase`.

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`([*n*])¶
Return 1 (or *n* if specified) bytes from a buffer without advancing the position. Only a single read on the raw stream is done to satisfy the call. The number of bytes returned may be less than requested since at most all the buffer's bytes from the current position to the end are returned.

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

`read1`(*n*)¶
Read and return up to *n* 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.

class `io.BufferedWriter`(*raw*[, *buffer_size*[, *max_buffer_size*]])¶

A buffer for a writeable sequential RawIO object. It inherits `BufferedIOBase`.

The constructor creates a `BufferedWriter` for the given writeable *raw* stream. If the *buffer_size* is not given, it defaults to `DEAFULT_BUFFER_SIZE`. If *max_buffer_size* is omitted, it defaults to twice the 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 or bytearray object, *b*, onto the raw stream and return the number of bytes written. A `BlockingIOError` is raised when the raw stream blocks.

class `io.BufferedRWPair`(*reader*, *writer*[, *buffer_size*[, *max_buffer_size*]])¶

A combined buffered writer and reader object for a raw stream that can be written to and read from. It has and supports both `read()`, `write()`, and their variants. This is useful for sockets and two-way pipes. 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`. The *max_buffer_size* (for the buffered writer) defaults to twice the buffer size.

`BufferedRWPair` implements all of `BufferedIOBase`'s methods.

class `io.BufferedRandom`(*raw*[, *buffer_size*[, *max_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`. The *max_buffer_size* (for the buffered writer) defaults to twice the buffer size.

`BufferedRandom` is capable of anything `BufferedReader` or `BufferedWriter` can do.

16.2.5. Text I/O¶

class `io.TextIOBase`¶

Base class for text streams. This class provides a character and line based interface to stream I/O. There is no `readinto()` method because Python's character strings are immutable. 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.

`newlines`¶
A string, a tuple of strings, or `None`, indicating the newlines translated so far.

`read`(*n*)¶
Read and return at most *n* characters from the stream as a single `str`. If *n* is negative or `None`, reads to EOF.

`readline`()¶
Read until newline or EOF and return a single `str`. If the stream is already at EOF, an empty string is returned.

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

class `io.TextIOWrapper`(*buffer*[, *encoding*[, *errors*[, *newline*[, *line_buffering*]]]])¶

A buffered text stream over a `BufferedIOBase` raw stream, *buffer*. It inherits `TextIOBase`.

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

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. When writing, `'xmlcharrefreplace'` (replace with the appropriate XML character reference) or `'backslashreplace'` (replace with backslashed escape sequences) can be used. Any other error handling name that has been registered with `codecs.register_error()` is also valid.

newline can be `None`, `''`, `'\n'`, `'\r'`, or `'\r\n'`. It controls the handling of line endings. If it is `None`, universal newlines is enabled. With this enabled, on input, the lines endings `'\n'`, `'\r'`, or `'\r\n'` are translated to `'\n'` before being returned to the caller. Conversely, on output, `'\n'` is translated to the system default line separator, `os.linesep`. If *newline* is any other of its legal values, that newline becomes the newline when the file is read and it is returned untranslated. On output, `'\n'` is converted to the *newline*.

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

`TextIOWrapper` provides these data attributes in addition to those of `TextIOBase` and its parents:

`errors`¶
The encoding and decoding error setting.

`line_buffering`¶
Whether line buffering is enabled.

class `io.StringIO`([*initial_value*[, *encoding*[, *errors*[, *newline*]]]])¶

An in-memory stream for text. It inherits `TextIOWrapper`.

Create a new StringIO stream with an inital value, encoding, error handling, and newline setting. See `TextIOWrapper`'s constructor for more information.

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

`getvalue`()¶
Return a `str` containing the entire contents of the buffer.

class `io.IncrementalNewlineDecoder`¶
A helper codec that decodes newlines for universal newlines mode. It inherits `codecs.IncrementalDecoder`.

Previous topic

16.1. `os` — Miscellaneous operating system interfaces

Next topic

16.3. `time` — Time access and conversions

This Page

- Show Source

Navigation

- index
- modules |
- next |

Last updated on Feb 26, 2010. Created using Sphinx 0.6.3.