Twatbot Documentation

Release 0.0.1

Simo Linkola

September 18, 2014

CONTENTS

Contents:

CHAPTER

ONE

TWATBOT

Twitter Bot for course work.

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

The io module provides the Python interfaces to stream handling. The builtin 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 raise 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 OS files.

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

Another IOBase subclass, TextIOBase, deals with the encoding and decoding of streams into text. TextIOWrapper, which extends it, is a buffered text interface to a buffered raw stream (*BufferedIOBase*). Finally, StringIO is a 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.

data:

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.

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

Exception raised when I/O would block on a non-blocking I/O stream

`io.`**`open`**`()`

Open file and return a stream. Raise IOError upon failure.

file is either a text or byte string giving the name (and the path if the file isn't in the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, 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; unneeded for 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 (appending ‘b’ to the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when ‘t’ is appended to 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 *io.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 passed. See the codecs module for the list of supported encodings.

errors is an optional string that specifies how encoding errors are to be handled—this argument should not be used in binary mode. 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.) See the documentation for codecs.register for a list of the permitted encoding error strings.

newline controls how universal newlines works (it only applies to text mode). It can be None, ‘’, ‘n’, ‘r’, and ‘rn’. 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 ‘rn’, 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, the underlying file descriptor will be kept open when the file is closed. This does not work when a file name is given and must be True in that case.

open() returns a file object whose type depends on the mode, and through which the standard file operations such as reading and writing are performed. 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 bytes a BytesIO can be used like a file opened in a binary mode.

class `io.`**`FileIO`**

file(name: str[, mode: str]) -> file IO object

Open a file. 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.

`close``()` → None. Close the file.
A closed file cannot be used for further I/O operations. close() may be called more than once without error. Changes the fileno to -1.

`closed`
True if the file is closed

`closefd`
True if the file descriptor will be closed

`fileno``()` → int. "file descriptor".
This is needed for lower-level file interfaces, such the fcntl module.

`isatty``()` → bool. True if the file is connected to a tty device.

`mode`
String giving the file mode

`read``(`*size: int*`)` → bytes. read at most size bytes, returned as bytes.
Only makes one system call, so less data may be returned than requested In non-blocking mode, returns None if no data is available. On end-of-file, returns ''.

`readable``()` → bool. True if file was opened in a read mode.

`readall``()` → bytes. read all data from the file, returned as bytes.
In non-blocking mode, returns as much as is immediately available, or None if no data is available. On end-of-file, returns ''.

`readinto``()` → Same as RawIOBase.readinto().

`seek``(`*offset: int*[, *whence: int*]`)` → None. Move to new file position.
Argument offset is a byte count. Optional argument whence defaults to 0 (offset from start of file, offset should be >= 0); other values are 1 (move relative to current position, positive or negative), and 2 (move relative to end of file, usually negative, although many platforms allow seeking beyond the end of a file). Note that not all file objects are seekable.

`seekable``()` → bool. True if file supports random-access.

`tell``()` → int. Current file position

`truncate``(`[*size: int*]`)` → None. Truncate the file to at most size bytes.
Size defaults to the current file position, as returned by tell().The current file position is changed to the value of size.

`writable``()` → bool. True if file was opened in a write mode.

`write``(`*b: bytes*`)` → int. Write bytes b to file, return number written.
Only makes one system call, so not all of the data may be written. The number of bytes actually written is returned.

class `io.`**`BytesIO`**

BytesIO([buffer]) -> object

Create a buffered I/O implementation using an in-memory bytes buffer, ready for reading and writing.

close() → None. Disable all I/O operations.

closed
True if the file is closed.

flush() → None. Does nothing.

getvalue() → bytes.
Retrieve the entire contents of the BytesIO object.

isatty() → False.
Always returns False since BytesIO objects are not connected to a tty-like device.

next
x.next() -> the next value, or raise StopIteration

read([*size*]) → read at most size bytes, returned as a string.
If the size argument is negative, read until EOF is reached. Return an empty string at EOF.

read1(*size*) → read at most size bytes, returned as a string.
If the size argument is negative or omitted, read until EOF is reached. Return an empty string at EOF.

readable() → bool. Returns True if the IO object can be read.

readinto(*bytearray*) → int. Read up to len(b) bytes into b.
Returns number of bytes read (0 for EOF), or None if the object is set not to block as has no data to read.

readline([*size*]) → next line from the file, as a string.
Retain newline. A non-negative size argument limits the maximum number of bytes to return (an incomplete line may be returned then). Return an empty string at EOF.

readlines([*size*]) → list of strings, each a line from the file.
Call readline() repeatedly and return a list of the lines so read. The optional size argument, if given, is an approximate bound on the total number of bytes in the lines returned.

seek(*pos*, *whence=0*) → int. Change stream position.

Seek to byte offset pos relative to position indicated by whence: 0 Start of stream (the default). pos should be >= 0; 1 Current position - pos may be negative; 2 End of stream - pos usually negative.

Returns the new absolute position.

seekable() → bool. Returns True if the IO object can be seeked.

tell() → current file position, an integer

truncate([*size*]) → int. Truncate the file to at most size bytes.
Size defaults to the current file position, as returned by tell(). The current file position is unchanged. Returns the new size.

writable() → bool. Returns True if the IO object can be written.

write(*bytes*) → int. Write bytes to file.
Return the number of bytes written.

writelines(*sequence_of_strings*) → None. Write strings to the file.
Note that newlines are not added. The sequence can be any iterable object producing strings. This is equivalent to calling write() for each string.

class io.**StringIO**
Text I/O implementation using an in-memory buffer.

The initial_value argument sets the value of object. The newline argument is like the one of TextIOWrapper's constructor.

close()
Close the IO object. Attempting any further operation after the object is closed will raise a ValueError.

This method has no effect if the file is already closed.

getvalue()
Retrieve the entire contents of the object.

next
x.next() -> the next value, or raise StopIteration

read()
Read at most n characters, returned as a string.

If the argument is negative or omitted, read until EOF is reached. Return an empty string at EOF.

readable() → bool. Returns True if the IO object can be read.

readline()
Read until newline or EOF.

Returns an empty string if EOF is hit immediately.

seek()
Change stream position.

Seek to character offset pos relative to position indicated by whence: 0 Start of stream (the default). pos should be >= 0; 1 Current position - pos must be 0; 2 End of stream - pos must be 0.

Returns the new absolute position.

seekable() → bool. Returns True if the IO object can be seeked.

tell()
Tell the current file position.

truncate()
Truncate size to pos.

The pos argument defaults to the current file position, as returned by tell(). The current file position is unchanged. Returns the new absolute position.

writable() → bool. Returns True if the IO object can be written.

write()
Write string to file.

Returns the number of characters written, which is always equal to the length of the string.

class io.**BufferedReader**
Create a new buffered reader using the given readable raw IO object.

next
x.next() -> the next value, or raise StopIteration

class io.**BufferedWriter**
A buffer for a writeable sequential RawIO object.

The constructor creates a BufferedWriter for the given writeable raw stream. If the buffer_size is not given, it defaults to DEFAULT_BUFFER_SIZE. max_buffer_size isn't used anymore.

class io.**BufferedRWPair**
A buffered reader and writer object together.

A buffered reader object and buffered writer object put together to form a sequential IO object that can read and write. This is typically used with a socket or two-way pipe.

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

class `io.`**`BufferedRandom`**

A buffered interface to random access streams.

The constructor creates a reader and writer for a seekable stream, raw, given in the first argument. If the buffer_size is omitted it defaults to DEFAULT_BUFFER_SIZE. max_buffer_size isn't used anymore.

`next`

x.next() -> the next value, or raise StopIteration

class `io.`**`TextIOWrapper`**

Character and line based layer over a BufferedIOBase object, buffer.

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

errors determines the strictness of encoding and decoding (see the codecs.register) and defaults to "strict".

newline controls how line endings are handled. It can be None, '', 'n', 'r', and 'rn'. 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 'rn', 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 line_buffering is True, a call to flush is implied when a call to write contains a newline character.

`next`

x.next() -> the next value, or raise StopIteration

PYTHON MODULE INDEX