# Python Chronicle Queue

## Introduction

This document describes the installation and use of the Python version of Chronicle Queue. Python Chronicle Queue is built directly on - and so is fully binary compatible with - the C++ version of Queue, which in turn is binary compatible with the Java version allowing complete interoperability between all three versions.

It should be noted that while the Python API is built on the fully-featured C++ version of Queue, a reduced, targeted subset only of the full Queue API is currently exposed through Python. This provides access to the main features to enable good functional overlap while keeping the API reasonably thin. If a required feature from C++ Queue is missing from the current API please contact Chronicle to discuss.

## Supported Platforms

Python Chronicle Queue is released as pre-built shared libraries. The following platforms are supported:

- x86 Linux
- x86 macOS
- ARM macOS

The Linux build has been directly tested across multiple distributions from kernel 2.6.32 upwards. The macOS builds have been directly tested on macOS 11 (Big Sur).

## Pre-Requisites

Python Chronicle Queue has the following dependencies:

- `python3` installed and available on the current `PATH`
- `libstdc++` usually available stand-alone, otherwise as part of a C++ compiler stack
- `Cython`, install using `pip3 install --upgrade cython`

## Installation

Python Chronicle Queue is packaged as a tar file containing the following:

- `builds/3.x` pre-built shared-libraries for different Python versions
- `Example.py` demo and test
- `install.sh` installation script (see below)
- `PrimitiveTest.py` demo and test of Primitives handling with bytes
- `Wire.py` demo and test of Wire API

To install, first unzip the tar file:

```
$> tar -zxvf python.chronicle.queue.tar.gz
```

Then run the install script:

```
$> ./install.sh
```

The install script sets up decorated links to the pre-built shared libraries (where the name decoration is required by `cython` which is used internally to wrap the C++ API), eg:

```
- cbytes.cpython-37m-darwin.so -> builds/3.7/libchronicle-base-python.so
- cqueue.cpython-37m-darwin.so -> builds/3.7libchronicle-queue-python.so
```

## License Key

Python Chronicle Queue requires a valid license key is set in the `CHRONICLE_KEY` environment variable. Keys can be obtained by contacting sales@chronicle.software

## Example

The installation pack includes `Example.py` which is a short demo and test, and which can be run as follows:

```
$> ./Example.py
```

or

```
$> PYTHONPATH=<path to links above> ./Example.py
```

# API

This section summarises the main interface for writing and reading native Python `bytes` or `bytearrays` to Chronicle Queue. The main queue interfaces follow a similar syntax to the Java and C++ versions.

## Imports

The following two imports are required, and reference the links created above available in either the current directory or via `PYTHONPATH`:

```
import cbytes
import cqueue
```

## Opening a Queue

Similarly to the Java and C++ interfaces, a `SingleChronicleQueueBuilder` is used to attach to a Queue, eg:

```
path = tempfile.TemporaryDirectory( None, None, "/tmp" )

q=cqueue.SingleChronicleQueueBuilder.\
   binary( path.name ).\
   blockSize(64 << 10).\
   rollCycle(cqueue.RollCycles.HOURLY()).\
   build()
```

The `binary()` argument sets the directory containing the queue files, and `blockSize()` the size of the memory-mapped segments. (Note: the example shows a test block size of only 64kB; in practice block sizes of several MB would be more typical).

The default roll cycle for a Python Chronicle Queue is `FAST_DAILY`, and this can be changed using the `rollCycle` option as illustrated above. The full set of roll cycles is listed below:

- `DAILY`
- `FAST_DAILY`
- `FAST_HOURLY`

- FOUR_HOURLY
- HALF_HOURLY
- HOURLY
- HUGE_DAILY
- HUGE_DAILY_XSPARSE
- LARGE_DAILY
- LARGE_HOURLY
- LARGE_HOURLY_SPARSE
- LARGE_HOURLY_XSPARSE
- MINUTELY
- SIX_HOURLY
- SMALL_DAILY
- TEN_MINUTELY
- TEST2_DAILY
- TEST4_DAILY
- TEST4_SECONDLY
- TEST8_DAILY
- TEST_DAILY
- TEST_HOURLY
- TEST_SECONDLY
- TWENTY_MINUTELY
- TWO_HOURLY
- XLARGE_DAILY

Finally `build()` creates and/or attaches to the Queue and returns a handle which is used further below.

## Appending to a Queue

Given a queue handle, `q`, an appender instance can be obtained by calling acquireAppender():

```
appender = q.acquireAppender()
```

The current Python Chronicle Queue API supports writing of Python `bytes` or `bytearrays` using the direct writeBytes function:

```
# write using python bytes or bytearray
def postOneMessage( appender, outbytes ):
   appender.writeBytes(outbytes)
```

The following complete example combines the above to illustrate writing both `bytes` and `bytearrays` given a handle, `q`, to a Chronicle Queue:

```
# write using python bytes or bytearray
def postOneMessage( appender, outbytes ):
   appender.writeBytes(outbytes)

def example():
   path = tempfile.TemporaryDirectory( None, None, "/tmp" )

   q=queue.SingleChronicleQueueBuilder.\
       binary( path.name ).blockSize(64 << 10).build()

   appender = q.acquireAppender()

   # write Python bytes
   outbytes = b"Hello World"
   postOneMessage( appender, outbytes )
```

```
   # write Python bytearray
   outbytearray = bytearray("Hello World", "utf8")
   postOneMessage( appender, outbytearray )
```

The index corresponding to the latest append to a queue for a given appender can be obtained as follows:

```
index = appender.lastIndexAppended()
```

## Tailing a Queue

Reading data from a Python Chronicle Queue uses a similar Tailer interface to the Java and C++ versions. Given a queue handle, `q`, a tailer instance can be obtained as follows:

```
tailer = q.createTailer()
```

Tailers use a scoped reading document context which uses Python's `with` statement to automatically manage resource acquisition and release (RAII), as follows:

```
with tailer.readingDocument() as context:
  # ...
```

Given a reading document context, a handle to the underlying Chronicle Bytes can be obtained by calling:

```
context.bytes()
```

The presence and type of data can be checked using the following:

```
context.isPresent()
context.isData()
context.isMetaData()
```

A copy of the data as Python `bytes` or `bytearray` can be obtained using the following respectively:

```
context.bytes().readToBytes()
context.bytes().readToByteArray()
```

Alternatively, a message can be read into an existing `bytearray` (cropping any data which exceeds the length of the `bytearray`). The length of data read is returned:

```
len = context.bytes().read(inbytearray)
```

The following complete example combines the above to illustrate reading both `bytes` and `bytearrays` given a Chronicle Queue handle, `q`:

```
# read message. return python bytes
def fetchOneMessage( tailer ):
   with tailer.readingDocument() as context:
       return b"" if not context.isPresent() else context.bytes().readToBytes()
```

```
# read message. return python bytearray
def fetchOneMessageArray( tailer ):
   with tailer.readingDocument() as context:
       return b"" if not context.isPresent() else context.bytes().readToByteArray()

# read message to given bytearray.
# no more than bytearray.size() bytes will be read. The rest will be discarded
def fetchOneMessageArray( tailer, inbytearray ):
   with tailer.readingDocument() as context:
       return 0 if not context.isPresent() else context.bytes().read(inbytearray)


def example():
   path = tempfile.TemporaryDirectory( None, None, "/tmp" )

   q=queue.SingleChronicleQueueBuilder.\
       binary( path.name ).blockSize(64 << 10).build()

   tailer = q.createTailer()

   inbytes = fetchOneMessage( tailer )
   inbytearray = fetchOneMessage( tailer )

   inbytearray = bytearray(100)
   len = fetchOneMessageArray( tailer, inbytearray )
```

The current tailer index or cycle can be obtained as follows:

```
index = tailer.index()
cycle = tailer.cycle()
```

and a tailer can be repositioned to a specific index using:

```
tailer.moveToIndex(index)
```

Similarly, a tailer can be positioned as the start or end of the queue using:

```
tailer.toStart()
tailer.toEnd()
```

## Chronicle Bytes

Alongside the Python `bytes` and `bytearray` interface above, Python Chronicle Queue also exposes part of the C++ Bytes API. As for Queue, only a small subset of the full Bytes API is currently exposed through Python while demand for Python features is determined. The API is likely to grow appreciably in the short term, and alongside this please contact sales@chronicle.software to discuss any specific requests for extending the API.

A Chronicle Bytes object is created using the `allocate` or `allocateElastic` functions, given a fixed-size or dynamic region respectively:

```
b1 = cbytes.allocate(size)          # fixed size
b2 = cbytes.allocateElastice(size) # dynamic size. Default 0
```

Individual POD-type data can be written or read from the `bytes` object using the following:

```
b.writeTYPE(data)          # append data at the current write position
b.writeTYPE(offset, data)  # write data at the given explicit offset

b.readTYPE()               # read data from the current read position
b.readTYPE(offset)         # read data from the given explicit offset
```

The POD types (`TYPE`) supported in the above are:

- `Boolean` # boolean, True/False
- `Byte` # 8-bit signed integer
- `UnsignedByte` # 8-bit unsigned integer
- `Short` # 16-bit signed integer
- `UnsignedShort` # 16-bit unsigned integer
- `Int24` # 24-bit integer
- `UnsignedInt24` # 24-bit unsigned integer
- `Int` # 32-bit signed integer
- `UnsignedInt` # 32-bit unsigned integer
- `Long` # 64-bit signed integer
- `UnsignedLong` # 64-bit unsigned integer
- `Float` # 32-bit floating point
- `Double` # 64-bit floating point

Compact encoding of natural 64-bit values using “stop bits” to truncate unnecessary serialisation bits can optionally be used instead:

```
b.writeStopBitLong(data)    # write stop-bit encoded 64-bit signed integer
b.writeStopBitDouble(data)  # write stop-bit encoded 64-bit floating point

b.readStopBitLong()         # read stop-bit encoded 64-bit signed integer (TODO)
b.readStopBitDouble()       # read stop-bit encoded 64-bit floating point (TODO)
```

Bytes can be written directly, and read as Python `bytes`, or `bytearray` as follows:

```
data = cbytes.allocate(8)
b.write(data)               # write stop-bit encoded 64-bit signed integer
out1 = b.readToBytes()      # read bytes; returns Python bytes
out2 = b.readToByteArray()  # read bytes; returns Python bytearray
out3 = b.read(array)        # read bytes to passed Python bytearray
```

Data can be added to the `bytes` in text format (as opposed to the default binary format) using the append, 8bit, and UTF8 methods, eg:

```
b.write8bit(string)           # write string as direct C-style 8-bit characters
b.writeUtf8(string)           # write UTF-8 encoded string
b.appendUtf8(string)          # append UTF-8 encoded string

b.appendChar(char)            # append single character
b.appendBoolean(bool)         # append 'T' or 'F' (true or false)
b.appendInt(int)              # append 32-bit integer as base-10
b.appendLong(long)            # append 64-bit integer as base-10
b.appendFloat(float)          # append 32-bit floating point, default encoding
b.appendDouble(double)        # append 64-bit floating point, default encoding
b.appendDouble2(double, int)  # append 64-bit floating point with given decimal places

b.read8bit()                  # read direct C-style 8-bit string, returns UTF-8
b.readUtf8()                  # reads UTF-8 encoded string, returns UTF-8
```

```
b.parseUtf8()                # read UTF8-encoded string up to stop character

b.parseBoolean()             # parse next character as boolean
b.parseInt()                 # parse base-10 32-bit integer
b.parseLong()                # parse base-10 64-bit integer
b.parseFloat()               # parse default-encoded 32-bit floating point
b.parseDouble()              # parse default-encoded 64-bit floating point
```

In addition, the contents of `bytes` can be written as text - either directly or using "`hexdump`" format using:

```
b.toString()
b.toHexString(size)          # use "hexdump" format. Truncate output after size bytes
```

For example:

```
def testTextPrimitive():
   b=cbytes.Bytes.allocateElastic()
   b.appendBoolean(True); b.appendChar('\n')
   b.appendInt(1); b.appendChar('\n')
   b.appendLong(2); b.appendChar('\n')
   b.appendChar('3'); b.appendChar('\n')
   b.appendFloat(4.1); b.appendChar('\n')
   b.appendDouble(5.2); b.appendChar('\n')
   b.appendDouble2(6.2999999, 3); b.appendChar('\n')

   print( b.toHexString() )
```

produces the following:

```
00000000 54 0a 31 0a 32 0a 33 0a  34 2e 31 0a 35 2e 32 0a T·1·2·3· 4.1·5.2·
00000010 36 2e 33 30 30 0a                               6.300·
```

The `bytes` can be cleared using:

```
b.clear()
```

while the current write position can be obtained using:

```
b.writePosition()
```

and the number of bytes available for reading can be obtained using:

```
b.readRemaining()
```

Chronicle Bytes also exposes a native static call to get the current nanosecond wallclock time as follows:

```
cbytes.Bytes.nanotime()
```

## Logging

Chronicle Python Queue supports the following hierarchy of log levels (from most to least verbose):

- `TRACE`
- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`
- `FATAL`
- `ALWAYS`

These levels are accessed through the LogLevel enum, with the application default set to `DEBUG`. For production instances it is recommended that the log level is set to `WARN`.

The current log level can be queried using the following:

```
cbytes.Log.level()
```

and the log level can be set using, eg:

```
cbytes.Log.level(cbytes.LogLevel.WARN)
```

## Chronicle Wire

As for `Bytes`, a subset of the full `Wire` API is currently exposed through Python while demand for the Python features is determined. The API is likely to grow in the short term, and alongside this please contact sales@chronicle.software to discuss any specific requests for extending the API.

A handle to Chronicle Wire is obtained from a `DocumentContext` (which in turn is obtained from an `Appender` or `Tailer` `writing/readingDocument` respectively). For example:

```
def example():
   path = tempfile.TemporaryDirectory( None, None, "/tmp" )

   q=queue.SingleChronicleQueueBuilder.\
       binary( path.name ).blockSize(64 << 10).build()

   appender = q.acquireAppender()
   with appender.writingDocument() as context:
      wire = context.wire()
      ...
```

Once a `wire` handle is obtained, data is written or read using a `ValueOut` or `ValueIn` handle respectively. Data can exist either standalone, or with an attached field, for example:

```
  ...
  wire = context.wire()
  valueOut1 = wire.write()           # get a standalone handle for writing
  valueOut2 = wire.write("key")      # get a handle for writing data to "key" field

  valueIn1 = wire.read()             # get a standalone handle for reading
  valueIn2 = wire.read("key")        # get a handle for reading data from "key" field
```

A given `ValueIn` or `ValueOut` handle can be reused multiple times under the same `DocumentContext`, eg:

```
  ...
  valueOut = wire.write()           # get a standalone handle for writing
  valueOut.text("hello world")      # write a string
  valueOut.int32(1234)              # write 32-bit signed integer

  ...
  valueIn = wire.read()             # get a standalone handle for reading
  text = valueIn.text()             # read string from wire
  i32 = valueIn.int32()             # read signed 32-bit integer
```

As a convenience, the `wire` handle associated with a `ValueIn` or `ValueOut` can be obtained as follows:

```
  ...
  wireIn = valueIn.wire()           # get handle to underlying wire for a ValueIn
  wireOut = valueOut.wire()         # get handle to underlying wire for a ValueOut
```

The full set of getters and setters using the `ValueOut` and `ValueIn` interfaces are listed below. Note the setter functions return a handle to the `ValueOut` object, enabling chained calls, eg:

```
  ...
  wire = context.wire()
  valueOut = wire.write()
  valueOut.text("hello world").int32(1234)
```

| ValueOut | ValueIn | Comment |
|---|---|---|
| `ValueOut ValueOut.text(string)`<br>`ValueOut ValueOut.writeString(string)` | `string ValueIn.text()`<br>`string ValueIn.readString()` | Arbitrary string |
| `ValueOut ValueOut.boolean(bool)` | `bool ValueIn.boolean()` | boolean |
| `ValueOut ValueOut.int8(int)`<br>`ValueOut ValueOut.writeByte(int)` | `int ValueIn.int8()`<br>`int ValueIn.readByte()` | 8-bit signed integer |
| `ValueOut ValueOut.uint8(int)` | `int ValueIn.uint8()` | 8-bit unsigned integer |
| `ValueOut ValueOut.int16(int)`<br>`ValueOut ValueOut.writeShort(int)` | `int ValueIn.int16()`<br>`int ValueIn.readShort()` | 16-bit signed integer |
| `ValueOut ValueOut.uint16(int)`<br>`ValueOut ValueOut.writeChar(int)` | `int ValueIn.uint16()`<br>`int ValueIn.readChar()` | 16-bit unsigned integer |
| `ValueOut ValueOut.int32(int)`<br>`ValueOut ValueOut.writeInt(int)` | `int ValueIn.int32()`<br>`int ValueIn.readInt()` | 32-bit signed integer |
| `ValueOut ValueOut.uint32(int)` | `int ValueIn.uint32()` | 32-bit unsigned integer |
| `ValueOut ValueOut.int64(int)`<br>`ValueOut ValueOut.writeLong(int)` | `int ValueIn.int64()`<br>`int ValueIn.readLong()` | 64-bit signed integer |
| `ValueOut ValueOut.float32(int)`<br>`ValueOut ValueOut.writeFloat(int)` | `float ValueIn.float32()`<br>`float ValueIn.readFloat()` | 32-bit floating point |
| `ValueOut ValueOut.float64(int)`<br>`ValueOut ValueOut.writeDouble(int)` | `float ValueIn.float64()`<br>`float ValueIn.readDouble()` | 64-bit floating point |
| `ValueOut ValueOut.bytes(bytes)` | `bytes ValueIn.bytes()` | Python `bytes` |
| `ValueOut ValueOut.byteArray(bytearray)` | `bytearray ValueIn.byteArray()` | Python `bytearray` |

Note: Chronicle Wire format uses reduced encoding where possible, so for example an `int64` does not necessarily always use the full 8 bytes in the underlying byte array.

A handle to the `Bytes` underlying a given `Wire` handle can be obtained from:

```
...
bytes = wire.bytes()              # get handle to underlying bytes for wire
```

For a ValueIn object, the availability of data to read can be tested with the following:

```
boolean ValueIn.hasNext()         # get handle to underlying bytes for wire
```

## Chronicle Queue

A handle to a Chronicle Queue is obtained using a `SingleChronicleQueueBuilder` (which creates the queue if it does not already exist), passing the following optional arguments:

```
q = cqueue.SingleChronicleQueueBuilder
    .binary(pathname)             # the pathname for the on-disk queue directory
    .blockSize(size)              # the segmentation blocksize (in bytes)
    .build()                      # builds the queue
```

Data is appended to the queue using an `Appender`, and read using a `Tailer`, which are obtained from a queue instance as follows:

```
appender = q.acquireAppender()
tailer = q.createTailer()
```

From an `appender`, data can be written directly using Chronicle Bytes, Python `bytes` or `bytearray`, or via a `DocumentContext` (see further below):

```
appender = q.acquireAppender()
appender.writeBytes(bytes)      # write bytes directly. The argument can be:
                                # - Chronicle Bytes (described above)
                                # - Python bytes
                                # - Python bytearray
appender.writingDocument()      # get scoped write context - see further below
```

The index for the most recently appended record can be obtained using:

```
appender.lastIndexAppended()
```

Given a `Tailer` instance, the following functions are available:

```
tailer = q.createTailer()
tailer.readingDocument()        # get scoped reading context - see further below
tailer.index()                  # get current read index
tailer.moveToIndex(index)       # reposition tailer to given index
tailer.toStart()                # move the tailer to the start of the queue
```

Both the `Appender` and `Tailer` provide scoped `DocumentContext` instances (`writingDocument()` and `readingDocument()` respectively). As well as providing various calls on the related document, this automatically manages resource acquisition and release (RAII) when used with Python's `with` statement, eg:

```
with appender.writingDocument() as context:
   # …
```

For a write document context, a write lock is held on the underlying queue for the duration of the `with` context. When the context `__exit__` method is (automatically) called at the end of the `with` block the data is written to the queue and the lock is released. If the write needs to be abandoned for any reason, calling `rollbackOnClose()` ensures no data is written to the queue when the context is released.

```
context.rollbackOnClose()
```

Given a `DocumentContext` the associated Chronicle Bytes can be obtained by calling:

```
context.bytes()
```

enabling the data to be read/written using the Chronicle Bytes interface as described in the previous section.

A `DocumentContext` object supports the following additional methods:

```
context.isMetaData()            # is this entry metadata (vs data)
context.isData()                # is this entry data (vs metadata)
context.metaData(bool)          # mark this entry as metadata (true) or data (false)
context.isPresent()             # is a record available at the current index
context.bytes()                 # access the underlying bytes (as above)
context.wire()                  # access the underlying wire handle (as above)
context.rollbackOnClose()       # undo the write operation (as above)
context.close()                 # explicitly close the context (normally with/scope)
```

For low-latency applications, `tailers` may spin repeatedly acquiring a `DocumentContext` and testing `isPresent()` to determine when new data is available. For Python Chronicle Queue a dedicated call is used to achieve the same from a `tailer` more efficiently without repeatedly crossing the C++/Python boundary:

```
context = tailer.waitReadingDocument() # busy wait for the next record
```

The above will either return the next available `DocumentContext` for reading, or block on the C++ side waiting for the next `DocumentContext` to be written by an `Appender`.

## Complete Example

For reference, the below shows a complete example of using Python for writing and reading data to Chronicle Queue.

```
#!/usr/bin/env python3

import tempfile
import cbytes
import cqueue

# write using python bytes or bytearray
def postOneMessage( appender, outbytes ):
   appender.writeBytes(outbytes)
#    with appender.writingDocument() as context:
#        context.bytes().write(outbytes)

# read message. return python bytes
def fetchOneMessage( tailer ):
   with tailer.readingDocument() as context:
       return b"" if not context.isPresent() else context.bytes().readToBytes()

# read message. return python bytearrau
def fetchOneMessageArray( tailer ):
   with tailer.readingDocument() as context:
       return b"" if not context.isPresent() else context.bytes().readToByteArray()

# read message to given bytearry.
# no more than bytearry.size() bytes will be read. The rest will be discarded
def fetchOneMessageArray( tailer, inbytearray ):
   with tailer.readingDocument() as context:
       return 0 if not context.isPresent() else context.bytes().read(inbytearray)

# test writing and reading python bytes
def testWriteBytes():
   path = tempfile.TemporaryDirectory( None, None, "/tmp" )

   q=cqueue.SingleChronicleQueueBuilder.binary( path.name ).blockSize(64 <<
10).rollCycle(cqueue.RollCycles.TEST_DAILY()).build()
   print( q.dumpMetastore() )

   appender = q.acquireAppender()
   tailer = q.createTailer()

   outbytes = b"Hello World"
   postOneMessage( appender, outbytes )

   inbytes = fetchOneMessage( tailer )

   print( inbytes )

   assert( inbytes == outbytes )

# test writing and reading python bytearray
def testWriteByteArray():
   path = tempfile.TemporaryDirectory( None, None, "/tmp" )

   q=cqueue.SingleChronicleQueueBuilder.binary( path.name ).blockSize(64 << 10).build()
   print( q.dumpMetastore() )

   appender = q.acquireAppender()
   tailer = q.createTailer()

   outbytearray = bytearray("Hello World", "utf8")
   postOneMessage( appender, outbytearray )
```

```
    inbytearray = fetchOneMessage( tailer )

    print( inbytearray )

    assert( inbytearray == outbytearray )

# test writing and reading into given bytearray
def testWriteByteArrayReference():
    path = tempfile.TemporaryDirectory( None, None, "/tmp" )

    q=cqueue.SingleChronicleQueueBuilder.binary( path.name ).blockSize(64 << 10).build()
    print( q.dumpMetastore() )

    appender = q.acquireAppender()
    tailer = q.createTailer()

    outbytearray = bytearray("Hello World", "utf8")
    postOneMessage( appender, outbytearray )

    inbytearray = bytearray(100)
    len = fetchOneMessageArray( tailer, inbytearray )

    print(len)
    print( inbytearray[:len] )

    assert( inbytearray[:len] == outbytearray )

# text read by index
def testExcerptIndex():
    path = tempfile.TemporaryDirectory( None, None, "/tmp" )

    q=cqueue.SingleChronicleQueueBuilder.binary( path.name ).blockSize(64 << 10).build()
    print( q.dumpMetastore() )

    appender = q.acquireAppender()
    tailer = q.createTailer()

    indexes = []

    for x in range(10):
        outbytearray = bytearray("Hello World: " + str(x), "utf8")
        postOneMessage( appender, outbytearray )
        index = appender.lastIndexAppended()
        indexes.append(index)
        print("LastAppendedIndex: " + str(index))

    # read all
    inbytearray = bytearray(100)
    for x in range(10):
        len = fetchOneMessageArray( tailer, inbytearray )
        print( inbytearray[:len] )
        expect = bytearray("Hello World: " + str(x), "utf8")
        assert( inbytearray[:len] == expect )

    # read by index
    expectItem = 0
    for index in indexes:
        print("Reading from index " + str(index))
        tailer.moveToIndex(index)
        len = fetchOneMessageArray( tailer, inbytearray )
        print( inbytearray[:len] )
        expect = bytearray("Hello World: " + str(expectItem), "utf8")
        expectItem += 1
        assert( inbytearray[:len] == expect )

if __name__ == '__main__':
```

```
    testWriteBytes()
    testWriteByteArray()
    testWriteByteArrayReference()
    testExcerptIndex()
```