qPython Documentation

Release 1.2.2

DEVnet

Mar 20, 2017

Contents

CHAPTER 1

Managing connection

qPython wraps connection to a q process in instances of the *QConnection* class.

```
q = qconnection.QConnection(host = 'localhost', port = 5000, username = 'tu',␣
↪password = 'secr3t', timeout = 3.0)
try:
    q.open()
    # ...
finally:
    q.close()
```

Note: the connection is not established when the connector instance is created. The connection is initialised explicitly by calling the *open()* method.

In order to terminate the remote connection, one has to invoke the *close()* method.

The *qconnection.QConnection* class provides a context manager API and can be used with a `with` statement:

```
with qconnection.QConnection(host = 'localhost', port = 5000) as q:
    print(q)
    print(q('{`int$ til x}', 10))
```

Types conversion configuration

Connection can be preconfigured to parse IPC messages according to a specified settings, e.g.: temporal vectors can be represented as raw vectors or converted to numpy *datetime64*/*timedelta64* representation.

```
# temporal values parsed to QTemporal and QTemporalList classes
q = qconnection.QConnection(host = 'localhost', port = 5000, numpy_temporals = False)

# temporal values parsed to numpy datetime64/timedelta64 arrays and atoms
q = qconnection.QConnection(host = 'localhost', port = 5000, numpy_temporals = True)
```

Conversion options can be also overwritten while executing synchronous/asynchronous queries (*sync()*, *async()*) or retrieving data from q (*receive()*).

Custom IPC protocol serializers/deserializers

Default IPC serializers (*QWriter* and _pandas.PandasQWriter) and deserializers (*QReader* and _pandas.PandasQReader) can be replaced with custom implementations. This allow users to override the default mapping between the q types and Python representation.

```
q = qconnection.QConnection(host = 'localhost', port = 5000, writer_class = MyQWriter,
↪ reader_class = MyQReader)
```

Refer to *Custom type mapping* for details.

CHAPTER 2

Queries

The *qPython* library supports both synchronous and asynchronous queries.

Synchronous query waits for service response and blocks requesting process until it receives the response. Asynchronous query does not block the requesting process and returns immediately without any result. The actual query result can be obtained either by issuing a corresponding request later on, or by setting up a listener to await and react accordingly to received data.

The *qPython* library provides following API methods in the *QConnection* class to interact with q:

- *sync()* - executes a synchronous query against the remote q service,
- *async()* - executes an asynchronous query against the remote q service,
- *query()* - executes a query against the remote q service.

These methods have following parameters:

- `query` is the definition of the query to be executed,
- `parameters` is a list of additional parameters used when executing given query.

In typical use case, `query` is the name of the function to call and `parameters` are its parameters. When parameters list is empty the query can be an arbitrary q expression (e.g. `0 +/ til 100`).

Synchronous queries

Executes a q expression:

```
>>> print(q.sync('til 10'))
[0 1 2 3 4 5 6 7 8 9]
```

Executes an anonymous q function with a single parameter:

```
>>> print(q.sync('{til x}', 10))
[0 1 2 3 4 5 6 7 8 9]
```

Executes an anonymous q function with two parameters:

```
>>> print(q.sync('{y + til x}', 10, 1))
[ 1  2  3  4  5  6  7  8  9 10]
>>> print(q.sync('{y + til x}', *[10, 1]))
[ 1  2  3  4  5  6  7  8  9 10]
```

The *QConnection* class implements the `__call__()` method. This allows *QConnection* instance to be called as a function:

```
>>> print(q('{y + til x}', 10, 1))
[ 1  2  3  4  5  6  7  8  9 10]
```

Asynchronous queries

Calls a anonymous function with a single parameter:

```
>>> q.async('{til x}', 10)
```

Executes a q expression:

```
>>> q.async('til 10')
```

Note: The asynchronous query doesn't fetch the result. Query result has to be retrieved explicitly.

In order to retrieve query result (for the *async()* or *query()* methods), one has to call:

- *receive()* method, which reads next message from the remote q service.

For example:

- Retrieves query result along with meta-information:

```
>>> q.query(qconnection.MessageType.SYNC,'{x}', 10)
>>> print(q.receive(data_only = False, raw = False))
QMessage: message type: 2, data size: 13, is_compressed: False, data: 10
```

- Retrieves parsed query result:

```
>>> q.query(qconnection.MessageType.SYNC,'{x}', 10)
>>> print(q.receive(data_only = True, raw = False))
10
```

```
>>> q.sync('asynchMult:{[a;b] res:a*b; (neg .z.w)(res) }')
>>> q.async('asynchMult', 2, 3)
>>> print(q.receive())
6
```

- Retrieves not-parsed (raw) query result:

```
>>> from binascii import hexlify
>>> q.query(qconnection.MessageType.SYNC,'{x}', 10)
>>> print(hexlify(q.receive(data_only = True, raw = True)))
fa0a000000
```

Type conversions configuration

Type conversion options can be overwritten while:

- executing synchronous query: *sync()*
- executing asynchronous query: *async()*
- retrieving data from q: *receive()*

These methods accepts the *options* keywords arguments:

```
>>> query = "{[x] 0Nd, `date$til x}"

>>> # retrieve function call as raw byte buffer
>>> from binascii import hexlify
>>> print(binascii.hexlify(q(query, 5, raw = True)))
0e000600000000000800000000010000000200000003000000040000000

>>> # perform a synchronous call and parse dates vector to numpy array
>>> print(q.sync(query, 5, numpy_temporals = True))
['NaT' '2000-01-01' '2000-01-02' '2000-01-03' '2000-01-04' '2000-01-05']

>>> # perform a synchronous call
>>> q.query(qconnection.MessageType.SYNC, query, 3)
>>> # retrieve query result and represent dates vector as raw data wrapped in␣
↪QTemporalList
>>> print(q.receive(numpy_temporals = False))
[NaT [metadata(qtype=-14)] 2000-01-01 [metadata(qtype=-14)]
 2000-01-02 [metadata(qtype=-14)] 2000-01-03 [metadata(qtype=-14)]]

>>> # serialize single element strings as q characters
>>> print(q.sync('{[x] type each x}', ['one', 'two', '3'], single_char_strings =␣
↪False))
[ 10,  10, -10]

>>> # serialize single element strings as q strings
>>> print(q.sync('{[x] type each x}', ['one', 'two', '3'], single_char_strings =␣
↪True))
[10, 10, 10]
```

CHAPTER 3

Types conversions

Data types supported by q and Python are incompatible and thus require additional translation. This page describes default rules used for converting data types between q and Python.

The translation mechanism used in qPython library is designed to:

- deserialized message from kdb+ can be serialized and send back to kdb+ without additional processing,
- end user can enforce type hinting for translation,
- efficient storage for tables and lists is backed with numpy arrays.

Default type mapping can be overriden by using custom IPC serializers or deserializers implementations.

Atoms

While parsing IPC message atom q types are translated to Python types according to this table:

q type	q num type	Python type
bool	-1	numpy.bool_
guid	-2	UUID
byte	-4	numpy.byte
short	-5	numpy.int16
integer	-6	numpy.int32
long	-7	numpy.int64
real	-8	numpy.float32
float	-9	numpy.float64
character	-10	single element str
timestamp	-12	QTemporal numpy.datetime64 ns
month	-13	QTemporal numpy.datetime64 M
date	-14	QTemporal numpy.datetime64 D
datetime	-15	QTemporal numpy.datetime64 ms
timespan	-16	QTemporal numpy.timedelta64 ns
minute	-17	QTemporal numpy.timedelta64 m
second	-18	QTemporal numpy.timedelta64 s
time	-19	QTemporal numpy.timedelta64 ms

Note: By default, temporal types in Python are represented as instances of *qtemporal.QTemporal* wrapping over numpy.datetime64 or numpy.timedelta64 with specified resolution. This setting can be modified (*numpy_temporals* = *True*) and temporal types can be represented without wrapping.

During the serialization to IPC protocol, Python types are mapped to q as described in the table:

Python type	q type	q num type
bool	bool	-1
—	byte	-4
—	short	-5
int	int	-6
long	long	-7
—	real	-8
double	float	-9
numpy.bool	bool	-1
numpy.byte	byte	-4
numpy.int16	short	-5
numpy.int32	int	-6
numpy.int64	long	-7
numpy.float32	real	-8
numpy.float64	float	-9
single element str	character	-10
QTemporal numpy.datetime64 ns	timestamp	-12
QTemporal numpy.datetime64 M	month	-13
QTemporal numpy.datetime64 D	date	-14
QTemporal numpy.datetime64 ms	datetime	-15
QTemporal numpy.timedelta64 ns	timespan	-16
QTemporal numpy.timedelta64 m	minute	-17
QTemporal numpy.timedelta64 s	second	-18
QTemporal numpy.timedelta64 ms	time	-19

Note: By default, single element strings are serialized as q characters. This setting can be modified (*single_char_strings* = *True*) and and single element strings are represented as q strings.

String and symbols

In order to distinguish symbols and strings on the Python side, following rules apply:

- q symbols are represented as `numpy.string_` type,
- q strings are mapped to plain Python strings in Python 2 and `bytes` in Python 3.

```
# Python 2
# `quickbrownfoxjumpsoveralazydog
<type 'numpy.string_'>
numpy.string_('quickbrownfoxjumpsoveralazydog')

# "quick brown fox jumps over a lazy dog"
<type 'str'>
'quick brown fox jumps over a lazy dog'

# Python 3
# `quickbrownfoxjumpsoveralazydog
<class 'numpy.bytes_'>
b'quickbrownfoxjumpsoveralazydog'

# "quick brown fox jumps over a lazy dog"
<class 'bytes'>
b'quick brown fox jumps over a lazy dog'
```

Note: By default, single element strings are serialized as q characters. This setting can be modified (*single_char_strings* = *True*) and and single element strings are represented as q strings.

```
>>> # serialize single element strings as q characters
>>> print(q.sync('{[x] type each x}', ['one', 'two', '3'], single_char_strings =␣
↪False))
[ 10,  10, -10]

>>> # serialize single element strings as q strings
>>> print(q.sync('{[x] type each x}', ['one', 'two', '3'], single_char_strings =␣
↪True))
[10, 10, 10]
```

Lists

qPython represents deserialized q lists as instances of *qcollection.QList* are mapped to *numpy* arrays.

```
# (0x01;0x02;0xff)
qlist(numpy.array([0x01, 0x02, 0xff], dtype=numpy.byte))

# <class 'qpython.qcollection.QList'>
# numpy.dtype: int8
# meta.qtype: -4
# str: [ 1  2 -1]
```

Generic lists are represented as a plain Python lists.

```
# (1;`bcd;"0bc";5.5e)
[numpy.int64(1), numpy.string_('bcd'), '0bc', numpy.float32(5.5)]
```

While serializing Python data to q following heuristic is applied:

- instances of *qcollection.QList* and *qcollection.QTemporalList* are serialized according to type indicator (`meta.qtype`):

```
qlist([1, 2, 3], qtype = QSHORT_LIST)
# (1h;2h;3h)

qlist([366, 121, qnull(QDATE)], qtype=QDATE_LIST)
# '2001.01.01 2000.05.01 0Nd'

qlist(numpy.array([uuid.UUID('8c680a01-5a49-5aab-5a65-d4bfddb6a661'),␣
↪qnull(QGUID)]), qtype=QGUID_LIST)
# ("G"$"8c680a01-5a49-5aab-5a65-d4bfddb6a661"; 0Ng)
```

- *numpy* arrays are serialized according to type of their *dtype* value:

```
numpy.array([1, 2, 3], dtype=numpy.int32)
# (1i;2i;3i)
```

- if *numpy* array *dtype* is not recognized by qPython, result q type is determined by type of the first element in the array,
- Python lists and tuples are represented as q generic lists:

```
[numpy.int64(42), None, numpy.string_('foo')]
(numpy.int64(42), None, numpy.string_('foo'))
# (42;::;`foo)
```

Note: *numpy* arrays with `dtype=='|S1'` are represented as atom character.

qPython provides an utility function *qcollection.qlist()* which simplifies creation of *qcollection.QList* and *qcollection.QTemporalList* instances.

The *qtype* module defines `QSTRING_LIST` const which simplifies creation of string lists:

```
qlist(numpy.array(['quick', 'brown', 'fox', 'jumps', 'over', 'a lazy', 'dog']), qtype␣
↪= QSTRING_LIST)
qlist(['quick', 'brown', 'fox', 'jumps', 'over', 'a lazy', 'dog'], qtype = QSTRING_
↪LIST)
['quick', 'brown', 'fox', 'jumps', 'over', 'a lazy', 'dog']
# ("quick"; "brown"; "fox"; "jumps"; "over"; "a lazy"; "dog")
```

Note: `QSTRING_LIST` type indicator indicates that list/array has to be mapped to q generic list.

Temporal lists

By default, lists of temporal values are represented as instances of *qcollection.QTemporalList* class. This class wraps the raw q representation of temporal data (i.e. `longs` for `timestamps`, `ints` for `months` etc.) and provides accessors which allow to convert raw data to `qcollection.QTemporal` instances in a lazy fashion.

```
>>> v = q.sync("2001.01.01 2000.05.01 0Nd", numpy_temporals = False)
>>> print('%s dtype: %s qtype: %d: %s' % (type(v), v.dtype, v.meta.qtype, v))
<class 'qpython.qcollection.QTemporalList'> dtype: int32 qtype: -14: [2001-01-01␣
↪[metadata(qtype=-14)] 2000-05-01 [metadata(qtype=-14)]
 NaT [metadata(qtype=-14)]]

>>> v = q.sync("2000.01.04D05:36:57.600 0Np", numpy_temporals = False)
>>> print('%s dtype: %s qtype: %d: %s' % (type(v), v.dtype, v.meta.qtype, v))
<class 'qpython.qcollection.QTemporalList'> dtype: int64 qtype: -12: [2000-01-
↪04T05:36:57.600000000+0100 [metadata(qtype=-12)]
 NaT [metadata(qtype=-12)]]
```

The IPC parser (*qreader.QReader*) can be instructed to represent the temporal vectors via *numpy.datetime64* or *numpy.timedelta64* arrays wrapped in *qcollection.QList* instances. The parsing option can be set either via *QConnection* constructor or as parameter to functions: (*sync()*) or (*receive()*).

```
>>> v = q.sync("2001.01.01 2000.05.01 0Nd", numpy_temporals = True)
>>> print('%s dtype: %s qtype: %d: %s' % (type(v), v.dtype, v.meta.qtype, v))
<class 'qpython.qcollection.QList'> dtype: datetime64[D] qtype: -14: ['2001-01-01'
↪'2000-05-01' 'NaT']

>>> v = q.sync("2000.01.04D05:36:57.600 0Np", numpy_temporals = True)
>>> print('%s dtype: %s qtype: %d: %s' % (type(v), v.dtype, v.meta.qtype, v))
<class 'qpython.qcollection.QList'> dtype: datetime64[ns] qtype: -12: ['2000-01-
↪04T05:36:57.600000000+0100' 'NaT']
```

In this parsing mode, temporal null values are converted to *numpy.NaT*.

The serialization mechanism (*qwriter.QWriter*) accepts both representations and doesn't require additional configuration.

There are two utility functions for conversions between both representations:

- The *qtemporal.array_to_raw_qtemporal()* function simplifies adjusting of *numpy.datetime64* or *numpy.timedelta64* arrays to q representation as raw integer vectors.
- The *qtemporal.array_from_raw_qtemporal()* converts raw temporal array to *numpy.datetime64* or *numpy.timedelta64* array.

Dictionaries

qPython represents q dictionaries with custom *qcollection.QDictionary* class.

Examples:

```
QDictionary(qlist(numpy.array([1, 2], dtype=numpy.int64), qtype=QLONG_LIST),
            qlist(numpy.array(['abc', 'cdefgh']), qtype = QSYMBOL_LIST))
# q: 1 2!`abc`cdefgh


QDictionary([numpy.int64(1), numpy.int16(2), numpy.float64(3.234), '4'],
            [numpy.string_('one'), qlist(numpy.array([2, 3]), qtype=QLONG_LIST), '456
↪', [numpy.int64(7), qlist(numpy.array([8, 9]), qtype=QLONG_LIST)]])
# q: (1;2h;3.234;"4")!(`one;2 3;"456";(7;8 9))
```

The *qcollection.QDictionary* class implements Python collection API.

Tables

The q tables are translated into custom `qcollection.QTable` class.

qPython provides an utility function *qcollection.qtable()* which simplifies creation of tables. This function also allow user to override default type conversions for each column and provide explicit q type hinting per column.

Examples:

```
qtable(qlist(numpy.array(['name', 'iq']), qtype = QSYMBOL_LIST),
      [qlist(numpy.array(['Dent', 'Beeblebrox', 'Prefect'])),
       qlist(numpy.array([98, 42, 126], dtype=numpy.int64))])

qtable(qlist(numpy.array(['name', 'iq']), qtype = QSYMBOL_LIST),
      [qlist(['Dent', 'Beeblebrox', 'Prefect'], qtype = QSYMBOL_LIST),
       qlist([98, 42, 126], qtype = QLONG_LIST)])

qtable(['name', 'iq'],
       [['Dent', 'Beeblebrox', 'Prefect'],
        [98, 42, 126]],
       name = QSYMBOL, iq = QLONG)

# flip `name`iq!(`Dent`Beeblebrox`Prefect;98 42 126)


qtable(('name', 'iq', 'fullname'),
       [qlist(numpy.array(['Dent', 'Beeblebrox', 'Prefect']), qtype = QSYMBOL_LIST),
        qlist(numpy.array([98, 42, 126]), qtype = QLONG_LIST),
        qlist(numpy.array(["Arthur Dent", "Zaphod Beeblebrox", "Ford Prefect"]),␣
↪qtype = QSTRING_LIST)])

# flip `name`iq`fullname!(`Dent`Beeblebrox`Prefect;98 42 126;("Arthur Dent"; "Zaphod␣
↪Beeblebrox"; "Ford Prefect"))
```

The keyed tables are represented by *qcollection.QKeyedTable* instances, where both keys and values are stored as a separate `qcollection.QTable` instances.

For example:

```
# ([eid:1001 1002 1003] pos:`d1`d2`d3;dates:(2001.01.01;2000.05.01;0Nd))
QKeyedTable(qtable(['eid'],
                   [qlist(numpy.array([1001, 1002, 1003]), qtype = QLONG_LIST)]),
            qtable(['pos', 'dates'],
                   [qlist(numpy.array(['d1', 'd2', 'd3']), qtype = QSYMBOL_LIST),
                    qlist(numpy.array([366, 121, qnull(QDATE)]), qtype = QDATE_
↪LIST)]))
```

Functions, lambdas and projections

IPC protocol type codes 100+ are used to represent functions, lambdas and projections. These types are represented as instances of base class *qtype.QFunction* or descendent classes:

- *qtype.QLambda* - represents q lambda expression, note the expression is required to be either:
  - q expression enclosed in {}, e.g.: `{x + y}`
  - k expression, e.g.: `k){x + y}`

- *qtype.QProjection* - represents function projection with parameters:

```
# { x + y}[3]
QProjection([QLambda('{x+y}'), numpy.int64(3)])
```

Note: Only *qtype.QLambda* and *qtype.QProjection* are serializable. qPython doesn't provide means to serialize other function types.

Errors

The q errors are represented as instances of *qtype.QException* class.

Null values

Please note that q `null` values are defined as:

```
_QNULL1 = numpy.int8(-2**7)
_QNULL2 = numpy.int16(-2**15)
_QNULL4 = numpy.int32(-2**31)
_QNULL8 = numpy.int64(-2**63)
_QNAN32 = numpy.fromstring('\x00\x00\xc0\x7f', dtype=numpy.float32)[0]
_QNAN64 = numpy.fromstring('\x00\x00\x00\x00\x00\x00\xf8\x7f', dtype=numpy.float64)[0]
_QNULL_BOOL = numpy.bool_(False)
_QNULL_SYM = numpy.string_('')
_QNULL_GUID = uuid.UUID('00000000-0000-0000-0000-000000000000')
```

Complete null mapping between q and Python is represented in the table:

q type	q null value	Python representation
bool	0b	_QNULL_BOOL
guid	0Ng	_QNULL_GUID
byte	0x00	_QNULL1
short	0Nh	_QNULL2
int	0N	_QNULL4
long	0Nj	_QNULL8
real	0Ne	_QNAN32
float	0n	_QNAN64
string	" "	' '
symbol	`	_QNULL_SYM
timestamp	0Np	_QNULL8
month	0Nm	_QNULL4
date	0Nd	_QNULL4
datetime	0Nz	_QNAN64
timespan	0Nn	_QNULL8
minute	0Nu	_QNULL4
second	0Nv	_QNULL4
time	0Nt	_QNULL4

The `qtype` provides two utility functions to work with null values:

- *qnull()* - retrieves null type for specified q type code,

- *is_null()* - checks whether value is considered a null for specified q type code.

Custom type mapping

Default type mapping can be overwritten by providing custom implementations of *QWriter* and/or *QReader* and proper initialization of the connection as described in *Custom IPC protocol serializers/deserializers*.

QWriter and *QReader* use parse time decorator (`Mapper`) which generates mapping between q and Python types. This mapping is stored in a static variable: `QReader._reader_map` and `QWriter._writer_map`. In case mapping is not found in the mapping:

- *QWriter* tries to find a matching qtype in the `~qtype.Q_TYPE` dictionary and serialize data as q atom,
- *QReader* tries to parse lists and atoms based on the type indicator in IPC stream.

While subclassing these classes, user can create copy of the mapping in the parent class and use parse time decorator:

```
class PandasQWriter(QWriter):
    _writer_map = dict.copy(QWriter._writer_map)     # create copy of default␣
↪serializer map
    serialize = Mapper(_writer_map)                  # upsert custom mapping

    @serialize(pandas.Series)
    def _write_pandas_series(self, data, qtype = None):
        # serialize pandas.Series into IPC stream
        # ..omitted for readability..
        self._write_list(data, qtype = qtype)


class PandasQReader(QReader):
    _reader_map = dict.copy(QReader._reader_map)    # create copy of default␣
↪deserializer map
    parse = Mapper(_reader_map)                     # overwrite default mapping

    @parse(QTABLE)
    def _read_table(self, qtype = QTABLE):
        # parse q table as pandas.DataFrame
        # ..omitted for readability..
        return pandas.DataFrame(data)
```

Refer to *Custom type IPC deserialization* for complete example.

CHAPTER 4

Pandas integration

The *qPython* allows user to use `pandas.DataFrame` and `pandas.Series` instead of `numpy.recarray` and `numpy.ndarray` to represent `q` tables and vectors.

In order to instrument *qPython* to use pandas data types user has to set `pandas` flag while:

- creating *`qconnection.QConnection`* instance,
- executing synchronous query: *`sync()`*,
- or retrieving data from q: *`receive()`*.

For example:

```
>>> with qconnection.QConnection(host = 'localhost', port = 5000, pandas = True) as q:
>>>     ds = q('(1i;0Ni;3i)', pandas = True)
>>>     print(ds)
0     1
1   NaN
2     3
dtype: float64
>>>     print(ds.meta)
metadata(qtype=6)

>>>     df =  q('flip `name`iq`fullname!(`Dent`Beeblebrox`Prefect;98 42 126;("Arthur␣
↪Dent"; "Zaphod Beeblebrox"; "Ford Prefect"))')
>>>     print(df)
         name   iq           fullname
0        Dent   98        Arthur Dent
1  Beeblebrox   42  Zaphod Beeblebrox
2     Prefect  126       Ford Prefect
>>>     print(df.meta)
metadata(iq=7, fullname=0, qtype=98, name=11)
>>>     print(q('type', df))
98

>>>     df =  q('([eid:1001 0N 1003;sym:`foo`bar`] pos:`d1`d2`d3;dates:(2001.01.01;
↪2000.05.01;0Nd))')
```

```
>>>     print(df)
         pos      dates
eid  sym
1001 foo  d1 2001-01-01
NaN  bar  d2 2000-05-01
1003      d3        NaT
>>>     print(df.meta)
metadata(dates=14, qtype=99, eid=7, sym=11, pos=11)
>>>     print(q('type', df))
99
```

Data conversions

If `pandas` flag is set, *qPython* converts the data according to following rules:

- `q` vectors are represented as `pandas.Series`:
  - `pandas.Series` is initialized with `numpy.ndarray` being result of parsing with `numpy_temporals` flag set to `True` (to ensure that temporal vectors are represented as numpy `datetime64`/`timedelta64` arrays).
  - q nulls are replaced with `numpy.NaN`. This can result in type promotion as described in pandas documentation.
  - `pandas.Series` is enriched with custom attribute `meta` (`qpython.MetaData`), which contains *qtype* of the vector. Note that this information is used while serializaing `pandas.Series` instance to IPC protocol.
- tables are represented as `pandas.DataFrame` instances:
  - individual columns are represented as `pandas.Series`.
  - `pandas.DataFrame` is enriched with custom attribute `meta` (`qpython.MetaData`), which lists *qtype* for each column in table. Note that this information is used during `pandas.DataFrame` serialization.
- keyed tables are backed as `pandas.DataFrame` instances as well:
  - index for `pandas.DataFrame` is created from key columns.
  - `pandas.DataFrame` is enriched with custom attribute `meta` (`qpython.MetaData`), which lists *qtype* for each column in table, including index ones. Note that this information is used during `pandas.DataFrame` serialization.

Type hinting

qPython applies following heuristic to determinate conversion between pandas and q types:

- `pandas.DataFrame` are serialized to q tables,
- `pandas.Series` are serialized to q lists according to these rules:
  - type of q list is determinate based on series *dtype*,
  - if mapping based on *dtype* is ambiguous (e.g. *dtype* is *object*), q type is determined by type of the first element in the array.

User can overwrite the default type mapping, by setting the `meta` attribute and provide additional information for the serializer.

Lists conversions

By default, series of `datetime64` is mapped to q timestamp:

```
pandas.Series(numpy.array([numpy.datetime64('2000-01-04T05:36:57.600Z', 'ms'), numpy.
↪datetime64('nat', 'ms')]))
# 2000.01.04D05:36:57.600000000 0N (type 12h)
```

`meta` attribute, can be used to change this and convert the series to, for example, q date list:

```
l = pandas.Series(numpy.array([numpy.datetime64('2000-01-04T05:36:57.600Z', 'ms'),␣
↪numpy.datetime64('nat', 'ms')]))
l.meta = MetaData(qtype = QDATE_LIST)
# 2000.01.04 0N (type 14h)
```

Similarly, the series of `float64` is mapped to q float (double precision) vector:

```
l = pandas.Series([1, numpy.nan, 3])
# 1 0n 3 (type 9h)
```

This can be overwritten to convert the list to integer vector:

```
l = pandas.Series([1, numpy.nan, 3])
l.meta = MetaData(qtype = QINT_LIST)
# 1 0N 3i (type 6h)
```

Table columns

Type hinting mechanism is useful for specifying the conversion rules for columns in the table. This can be used either to enforce the type conversions or provide information for ambiguous mappings.

```
t = pandas.DataFrame(OrderedDict((('pos', pandas.Series(['A', 'B', 'C'])),
                                  ('dates', pandas.Series(numpy.array([numpy.
↪datetime64('2001-01-01'), numpy.datetime64('2000-05-01'), numpy.datetime64('NaT')],␣
↪dtype='datetime64[D]')))))))

# pos dates
# ---------------------------------
# A   2001.01.01D00:00:00.000000000
# B   2000.05.01D00:00:00.000000000
# C
#
# meta:
# c    | t f a
# -----| -----
# pos  | c
# dates| p

t = pandas.DataFrame(OrderedDict((('pos', pandas.Series(['A', 'B', 'C'])),
                                  ('dates', pandas.Series(numpy.array([numpy.
↪datetime64('2001-01-01'), numpy.datetime64('2000-05-01'), numpy.datetime64('NaT')],␣
↪dtype='datetime64[D]')))))))
```

```
t.meta = MetaData(pos = QSYMBOL_LIST, dates = QDATE_LIST)

# pos dates
# --------------
# A   2001.01.01
# B   2000.05.01
# C
#
# meta:
# c    | t f a
# -----| -----
# pos  | s
# dates| d
```

Keyed tables

By default, `pandas.DataFrame` is represented as a q table. During the serialization index information is discarded:

```
t = pandas.DataFrame(OrderedDict((('eid', pandas.Series(numpy.array([1001, 1002,␣
↪1003])))),
                                 ('pos', pandas.Series(numpy.array(['d1', 'd2', 'd3
↪'])))),
                                 ('dates', pandas.Series(numpy.array([numpy.
↪datetime64('2001-01-01'), numpy.datetime64('2000-05-01'), numpy.datetime64('NaT')],␣
↪dtype='datetime64[D]'))))))
t.reset_index(drop = True)
t.set_index(['eid'], inplace = True)
t.meta = MetaData(pos = QSYMBOL_LIST, dates = QDATE_LIST)

# pos dates
# --------------
# d1  2001.01.01
# d2  2000.05.01
# d3
#
# meta:
# c    | t f a
# -----| -----
# pos  | s
# dates| d
```

In order to preserve the index data and represent `pandas.DataFrame` as a q keyed table, use type hinting mechanism to enforce the serialization rules:

```
t = pandas.DataFrame(OrderedDict((('eid', pandas.Series(numpy.array([1001, 1002,␣
↪1003])))),
                                 ('pos', pandas.Series(numpy.array(['d1', 'd2', 'd3
↪'])))),
                                 ('dates', pandas.Series(numpy.array([numpy.
↪datetime64('2001-01-01'), numpy.datetime64('2000-05-01'), numpy.datetime64('NaT')],␣
↪dtype='datetime64[D]'))))))
t.reset_index(drop = True)
t.set_index(['eid'], inplace = True)
t.meta = MetaData(pos = QSYMBOL_LIST, dates = QDATE_LIST, qtype = QKEYED_TABLE)

# eid | pos dates
```

```
# ----| --------------
# 1001| d1  2001.01.01
# 1002| d2  2000.05.01
# 1003| d3
#
# meta:
# c    | t f a
# -----| -----
# eid  | j
# pos  | s
# dates| d
```

CHAPTER 5

Usage examples

Synchronous query

Following example presents how to execute simple, synchronous query against a remote q process:

```
from qpython import qconnection


if __name__ == '__main__':
    # create connection object
    q = qconnection.QConnection(host='localhost', port=5000)
    # initialize connection
    q.open()

    print(q)
    print('IPC version: %s. Is connected: %s' % (q.protocol_version, q.is_
↪connected()))

    # simple query execution via: QConnection.__call__
    data = q('{`int$ til x}', 10)
    print('type: %s, numpy.dtype: %s, meta.qtype: %s, data: %s ' % (type(data), data.
↪dtype, data.meta.qtype, data))

    # simple query execution via: QConnection.sync
    data = q.sync('{`long$ til x}', 10)
    print('type: %s, numpy.dtype: %s, meta.qtype: %s, data: %s ' % (type(data), data.
↪dtype, data.meta.qtype, data))

    # low-level query and read
    q.query(qconnection.MessageType.SYNC, '{`short$ til x}', 10) # sends a SYNC query
    msg = q.receive(data_only=False, raw=False) # retrieve entire message
    print('type: %s, message type: %s, data size: %s, is_compressed: %s ' %␣
↪(type(msg), msg.type, msg.size, msg.is_compressed))
    data = msg.data
    print('type: %s, numpy.dtype: %s, meta.qtype: %s, data: %s ' % (type(data), data.
↪dtype, data.meta.qtype, data))
```

```
    # close connection
    q.close()
```

This code prints to the console:

```
:localhost:5000
IPC version: 3. Is connected: True
type: <class 'qpython.qcollection.QList'>, numpy.dtype: int32, meta.qtype: 6, data:␣
↪[0 1 2 3 4 5 6 7 8 9]
type: <class 'qpython.qcollection.QList'>, numpy.dtype: int64, meta.qtype: 7, data:␣
↪[0 1 2 3 4 5 6 7 8 9]
type: <class 'qpython.qreader.QMessage'>, message type: 2, data size: 34, is_
↪compressed: False
type: <class 'qpython.qcollection.QList'>, numpy.dtype: int16, meta.qtype: 5, data:␣
↪[0 1 2 3 4 5 6 7 8 9]
```

Asynchronous query

Following example presents how to execute simple, asynchronous query against a remote q process:

```
import random
import threading
import time

from qpython import qconnection
from qpython.qtype import QException
from qpython.qconnection import MessageType
from qpython.qcollection import QDictionary


class ListenerThread(threading.Thread):

    def __init__(self, q):
        super(ListenerThread, self).__init__()
        self.q = q
        self._stopper = threading.Event()

    def stop(self):
        self._stopper.set()

    def stopped(self):
        return self._stopper.isSet()

    def run(self):
        while not self.stopped():
            print('.')
            try:
                message = self.q.receive(data_only = False, raw = False) # retrieve␣
↪entire message

                if message.type != MessageType.ASYNC:
                    print('Unexpected message, expected message of type: ASYNC')

                print('type: %s, message type: %s, data size: %s, is_compressed: %s '
↪% (type(message), message.type, message.size, message.is_compressed))
```

```
                print(message.data)

                if isinstance(message.data, QDictionary):
                    # stop after 10th query
                    if message.data[b'queryid'] == 9:
                        self.stop()

            except QException as e:
                print(e)


if __name__ == '__main__':
    # create connection object
    q = qconnection.QConnection(host = 'localhost', port = 5000)
    # initialize connection
    q.open()

    print(q)
    print('IPC version: %s. Is connected: %s' % (q.protocol_version, q.is_
↪connected()))

    try:
        # definition of asynchronous multiply function
        # queryid - unique identifier of function call - used to identify
        # the result
        # a, b - parameters to the query
        q.sync('asynchMult:{[queryid;a;b] res:a*b; (neg .z.w)(`queryid`result!
↪(queryid;res)) }');

        t = ListenerThread(q)
        t.start()

        for x in range(10):
            a = random.randint(1, 100)
            b = random.randint(1, 100)
            print('Asynchronous call with queryid=%s with arguments: %s, %s' % (x, a,␣
↪b))
            q.async('asynchMult', x, a, b);

        time.sleep(1)
    finally:
        q.close()
```

Interactive console

This example depicts how to create a simple interactive console for communication with a q process:

```
import qpython
from qpython import qconnection
from qpython.qtype import QException

try:
    input = raw_input
except NameError:
    pass
```

```
if __name__ == '__main__':
    print('qPython %s Cython extensions enabled: %s' % (qpython.__version__, qpython._
↪_is_cython_enabled__))
    with qconnection.QConnection(host = 'localhost', port = 5000) as q:
        print(q)
        print('IPC version: %s. Is connected: %s' % (q.protocol_version, q.is_
↪connected()))

        while True:
            try:
                x = input('Q)')
            except EOFError:
                print('')
                break

            if x == '\\\\':
                break

            try:
                result = q(x)
                print(type(result))
                print(result)
            except QException as msg:
                print('q error: \'%s' % msg)
```

Twisted integration

This example presents how the *qPython* can be used along with Twisted to build asynchronous client:

Note: This sample code overwrites *.u.sub* and *.z.ts* functions on q process.

```
import struct
import sys

from twisted.internet.protocol import Protocol, ClientFactory

from twisted.internet import reactor
from qpython.qconnection import MessageType, QAuthenticationException
from qpython.qreader import QReader
from qpython.qwriter import QWriter, QWriterException



class IPCProtocol(Protocol):

    class State(object):
        UNKNOWN = -1
        HANDSHAKE = 0
        CONNECTED = 1

    def connectionMade(self):
        self.state = IPCProtocol.State.UNKNOWN
```

```
        self.credentials = self.factory.username + ':' + self.factory.password if␣
↪self.factory.password else ''

        self.transport.write(self.credentials + '\3\0')

        self._message = None

    def dataReceived(self, data):
        if self.state == IPCProtocol.State.CONNECTED:
            try:
                if not self._message:
                    self._message = self._reader.read_header(source=data)
                    self._buffer = ''

                self._buffer += data
                buffer_len = len(self._buffer) if self._buffer else 0

                while self._message and self._message.size <= buffer_len:
                    complete_message = self._buffer[:self._message.size]

                    if buffer_len > self._message.size:
                        self._buffer = self._buffer[self._message.size:]
                        buffer_len = len(self._buffer) if self._buffer else 0
                        self._message = self._reader.read_header(source=self._buffer)
                    else:
                        self._message = None
                        self._buffer = ''
                        buffer_len = 0

                    self.factory.onMessage(self._reader.read(source=complete_message,␣
↪numpy_temporals=True))
            except:
                self.factory.onError(sys.exc_info())
                self._message = None
                self._buffer = ''

        elif self.state == IPCProtocol.State.UNKNOWN:
            # handshake
            if len(data) == 1:
                self._init(data)
            else:
                self.state = IPCProtocol.State.HANDSHAKE
                self.transport.write(self.credentials + '\0')

        else:
            # protocol version fallback
            if len(data) == 1:
                self._init(data)
            else:
                raise QAuthenticationException('Connection denied.')

    def _init(self, data):
        self.state = IPCProtocol.State.CONNECTED
        self.protocol_version = min(struct.unpack('B', data)[0], 3)
        self._writer = QWriter(stream=None, protocol_version=self.protocol_version)
        self._reader = QReader(stream=None)

        self.factory.clientReady(self)
```

```
    def query(self, msg_type, query, *parameters):
        if parameters and len(parameters) > 8:
            raise QWriterException('Too many parameters.')

        if not parameters or len(parameters) == 0:
            self.transport.write(self._writer.write(query, msg_type))
        else:
            self.transport.write(self._writer.write([query] + list(parameters), msg_
↪type))



class IPCClientFactory(ClientFactory):

    protocol = IPCProtocol

    def __init__(self, username, password, connect_success_callback, connect_fail_
↪callback, data_callback, error_callback):
        self.username = username
        self.password = password
        self.client = None

        # register callbacks
        self.connect_success_callback = connect_success_callback
        self.connect_fail_callback = connect_fail_callback
        self.data_callback = data_callback
        self.error_callback = error_callback


    def clientConnectionLost(self, connector, reason):
        print('Lost connection.  Reason: %s' % reason)
        # connector.connect()

    def clientConnectionFailed(self, connector, reason):
        if self.connect_fail_callback:
            self.connect_fail_callback(self, reason)

    def clientReady(self, client):
        self.client = client
        if self.connect_success_callback:
            self.connect_success_callback(self)

    def onMessage(self, message):
        if self.data_callback:
            self.data_callback(self, message)

    def onError(self, error):
        if self.error_callback:
            self.error_callback(self, error)

    def query(self, msg_type, query, *parameters):
        if self.client:
            self.client.query(msg_type, query, *parameters)



def onConnectSuccess(source):
```

```
    print('Connected, protocol version: %s' % source.client.protocol_version)
    source.query(MessageType.SYNC, '.z.ts:{(handle)(`timestamp$100?
↪1000000000000000000)}')
    source.query(MessageType.SYNC, '.u.sub:{[t;s] handle:: neg .z.w}')
    source.query(MessageType.ASYNC, '.u.sub', 'trade', '')


def onConnectFail(source, reason):
    print('Connection refused: %s' % reason)


def onMessage(source, message):
    print('Received: %s %s' % (message.type, message.data))


def onError(source, error):
    print('Error: %s' % error)


if __name__ == '__main__':
    factory = IPCClientFactory('user', 'pwd', onConnectSuccess, onConnectFail,␣
↪onMessage, onError)
    reactor.connectTCP('localhost', 5000, factory)
    reactor.run()
```

Subscribing to tick service

This example depicts how to subscribe to standard kdb+ tickerplant service:

```
import numpy
import threading
import sys

from qpython import qconnection
from qpython.qtype import QException
from qpython.qconnection import MessageType
from qpython.qcollection import QTable


class ListenerThread(threading.Thread):

    def __init__(self, q):
        super(ListenerThread, self).__init__()
        self.q = q
        self._stopper = threading.Event()

    def stopit(self):
        self._stopper.set()

    def stopped(self):
        return self._stopper.is_set()

    def run(self):
        while not self.stopped():
            print('.')
```

```
            try:
                message = self.q.receive(data_only = False, raw = False) # retrieve␣
↪entire message

                if message.type != MessageType.ASYNC:
                    print('Unexpected message, expected message of type: ASYNC')

                print('type: %s, message type: %s, data size: %s, is_compressed: %s '
↪% (type(message), message.type, message.size, message.is_compressed))

                if isinstance(message.data, list):
                    # unpack upd message
                    if len(message.data) == 3 and message.data[0] == b'upd' and␣
↪isinstance(message.data[2], QTable):
                        for row in message.data[2]:
                            print(row)

            except QException as e:
                print(e)


if __name__ == '__main__':
    with qconnection.QConnection(host = 'localhost', port = 17010) as q:
        print(q)
        print('IPC version: %s. Is connected: %s' % (q.protocol_version, q.is_
↪connected()))
        print('Press <ENTER> to close application')

        # subscribe to tick
        response = q.sync('.u.sub', numpy.string_('trade'), numpy.string_(''))
        # get table model
        if isinstance(response[1], QTable):
            print('%s table data model: %s' % (response[0], response[1].dtype))

        t = ListenerThread(q)
        t.start()

        sys.stdin.readline()

        t.stopit()
```

Data publisher

This example shows how to stream data to the kdb+ process using standard tickerplant API:

```
import datetime
import numpy
import random
import threading
import sys
import time

from qpython import qconnection
from qpython.qcollection import qlist
from qpython.qtype import QException, QTIME_LIST, QSYMBOL_LIST, QFLOAT_LIST
```

```
class PublisherThread(threading.Thread):

    def __init__(self, q):
        super(PublisherThread, self).__init__()
        self.q = q
        self._stopper = threading.Event()

    def stop(self):
        self._stopper.set()

    def stopped(self):
        return self._stopper.isSet()

    def run(self):
        while not self.stopped():
            print('.')
            try:
                # publish data to tick
                # function: .u.upd
                # table: ask
                self.q.sync('.u.upd', numpy.string_('ask'), self.get_ask_data())

                time.sleep(1)
            except QException as e:
                print(e)
            except:
                self.stop()

    def get_ask_data(self):
        c = random.randint(1, 10)

        today = numpy.datetime64(datetime.datetime.now().replace(hour=0, minute=0,␣
↪second=0, microsecond=0))

        time = [numpy.timedelta64((numpy.datetime64(datetime.datetime.now()) - today),
↪ 'ms') for x in range(c)]
        instr = ['instr_%d' % random.randint(1, 100) for x in range(c)]
        src = ['qPython' for x in range(c)]
        ask = [random.random() * random.randint(1, 100) for x in range(c)]

        data = [qlist(time, qtype=QTIME_LIST), qlist(instr, qtype=QSYMBOL_LIST),␣
↪qlist(src, qtype=QSYMBOL_LIST), qlist(ask, qtype=QFLOAT_LIST)]
        print(data)
        return data


if __name__ == '__main__':
    with qconnection.QConnection(host='localhost', port=17010) as q:
        print(q)
        print('IPC version: %s. Is connected: %s' % (q.protocol_version, q.is_
↪connected()))
        print('Press <ENTER> to close application')

        t = PublisherThread(q)
        t.start()
```

```
        sys.stdin.readline()

        t.stop()
        t.join()
```

Custom type IPC deserialization

This example shows how to override standard deserialization type mapping with two different *QReader* sub-classes. Please refer to *Custom type mapping* on implementation aspects:

```
import numpy

from qpython import qconnection
from qpython.qreader import QReader
from qpython.qtype import QSYMBOL, QSYMBOL_LIST, Mapper


class StringQReader(QReader):
    # QReader and QWriter use decorators to map data types and corresponding function␣
↪handlers
    _reader_map = dict.copy(QReader._reader_map)
    parse = Mapper(_reader_map)

    def _read_list(self, qtype):
        if qtype == QSYMBOL_LIST:
            self._buffer.skip()
            length = self._buffer.get_int()
            symbols = self._buffer.get_symbols(length)
            return [s.decode(self._encoding) for s in symbols]
        else:
            return QReader._read_list(self, qtype = qtype)

    @parse(QSYMBOL)
    def _read_symbol(self, qtype = QSYMBOL):
        return numpy.string_(self._buffer.get_symbol()).decode(self._encoding)



class ReverseStringQReader(QReader):
    # QReader and QWriter use decorators to map data types and corresponding function␣
↪handlers
    _reader_map = dict.copy(QReader._reader_map)
    parse = Mapper(_reader_map)

    @parse(QSYMBOL_LIST)
    def _read_symbol_list(self, qtype):
        self._buffer.skip()
        length = self._buffer.get_int()
        symbols = self._buffer.get_symbols(length)
        return [s.decode(self._encoding)[::-1] for s in symbols]

    @parse(QSYMBOL)
    def _read_symbol(self, qtype = QSYMBOL):
        return numpy.string_(self._buffer.get_symbol()).decode(self._encoding)[::-1]
```

```
if __name__ == '__main__':
    with qconnection.QConnection(host = 'localhost', port = 5000, reader_class =␣
↪StringQReader) as q:
        symbols = q.sync('`foo`bar')
        print(symbols, type(symbols), type(symbols[0]))

        symbol = q.sync('`foo')
        print(symbol, type(symbol))


    with qconnection.QConnection(host = 'localhost', port = 5000, reader_class =␣
↪ReverseStringQReader) as q:
        symbols = q.sync('`foo`bar')
        print(symbols, type(symbols), type(symbols[0]))

        symbol = q.sync('`foo')
        print(symbol, type(symbol))
```

CHAPTER 6

API documentation:

qpython package

qpython.qconnection module

exception qpython.qconnection.**QConnectionException**
Bases: exceptions.Exception

Raised when a connection to the q service cannot be established.

exception qpython.qconnection.**QAuthenticationException**
Bases: *qpython.qconnection.QConnectionException*

Raised when a connection to the q service is denied.

class qpython.qconnection.**MessageType**
Bases: object

Enumeration defining IPC protocol message types.

ASYNC = 0

SYNC = 1

RESPONSE = 2

class qpython.qconnection.**QConnection**(*host*, *port*, *username=None*, *password=None*, *timeout=None*, *encoding='latin-1'*, *reader_class=None*, *writer_class=None*, ***options*)
Bases: object

Connector class for interfacing with the q service.

Provides methods for synchronous and asynchronous interaction.

The *QConnection* class provides a context manager API and can be used with a with statement:

```
with qconnection.QConnection(host = 'localhost', port = 5000) as q:
    print(q)
    print(q('{`int$ til x}', 10))
```

Parameters

- *host* (*string*) - q service hostname
- *port* (*integer*) - q service port
- *username* (*string* or *None*) - username for q authentication/authorization
- *password* (*string* or *None*) - password for q authentication/authorization
- *timeout* (*nonnegative float* or *None*) - set a timeout on blocking socket operations
- *encoding* (*string*) - string encoding for data deserialization
- *reader_class* (subclass of *QReader*) - data deserializer
- *writer_class* (subclass of *QWriter*) - data serializer

Options

- *raw* (*boolean*) - if `True` returns raw data chunk instead of parsed data, **Default**: `False`
- *numpy_temporals* (*boolean*) - if `False` temporal vectors are backed by raw q representation (*QTemporalList*, *QTemporal*) instances, otherwise are represented as *numpy datetime64/timedelta64* arrays and atoms, **Default**: `False`
- *single_char_strings* (*boolean*) - if `True` single char Python strings are encoded as q strings instead of chars, **Default**: `False`

protocol_version

Retrieves established version of the IPC protocol.

Returns *integer* – version of the IPC protocol

open()

Initialises connection to q service.

If the connection hasn't been initialised yet, invoking the `open()` creates a new socket and performs a handshake with a q service.

Raises *QConnectionException*, *QAuthenticationException*

close()

Closes connection with the q service.

is_connected()

Checks whether connection with a q service has been established.

Connection is considered inactive when:

- it has not been initialised,
- it has been closed.

Returns *boolean* – `True` if connection has been established, `False` otherwise

query(*msg_type*, *query*, **parameters*, ***options*)

Performs a query against a q service.

In typical use case, *query* is the name of the function to call and *parameters* are its parameters. When *parameters* list is empty, the query can be an arbitrary q expression (e.g. `0 +/ til 100`).

Calls a anonymous function with a single parameter:

```
>>> q.query(qconnection.MessageType.SYNC,'{til x}', 10)
```

Executes a q expression:

```
>>> q.query(qconnection.MessageType.SYNC,'til 10')
```

Parameters

- *msg_type* (one of the constants defined in *MessageType*) - type of the query to be executed
- *query* (*string*) - query to be executed
- *parameters* (*list* or *None*) - parameters for the query

Options

- *single_char_strings* (*boolean*) - if `True` single char Python strings are encoded as q strings instead of chars, **Default**: `False`

Raises *QConnectionException*, *QWriterException*

sync(*query*, **parameters*, ***options*)

Performs a synchronous query against a q service and returns parsed data.

In typical use case, *query* is the name of the function to call and *parameters* are its parameters. When *parameters* list is empty, the query can be an arbitrary q expression (e.g. `0 +/ til 100`).

Executes a q expression:

```
>>> print(q.sync('til 10'))
[0 1 2 3 4 5 6 7 8 9]
```

Executes an anonymous q function with a single parameter:

```
>>> print(q.sync('{til x}', 10))
[0 1 2 3 4 5 6 7 8 9]
```

Executes an anonymous q function with two parameters:

```
>>> print(q.sync('{y + til x}', 10, 1))
[ 1  2  3  4  5  6  7  8  9 10]
```

```
>>> print(q.sync('{y + til x}', *[10, 1]))
[ 1  2  3  4  5  6  7  8  9 10]
```

The `sync()` is called from the overloaded `__call__()` function. This allows *QConnection* instance to be called as a function:

```
>>> print(q('{y + til x}', 10, 1))
[ 1  2  3  4  5  6  7  8  9 10]
```

Parameters

- *query* (*string*) - query to be executed

- *parameters* (*list* or *None*) - parameters for the query

Options

- *raw* (*boolean*) - if `True` returns raw data chunk instead of parsed data, **Default**: `False`
- *numpy_temporals* (*boolean*) - if `False` temporal vectors are backed by raw q representation (*QTemporalList*, *QTemporal*) instances, otherwise are represented as *numpy datetime64/timedelta64* arrays and atoms, **Default**: `False`
- *single_char_strings* (*boolean*) - if `True` single char Python strings are encoded as q strings instead of chars, **Default**: `False`

Returns query result parsed to Python data structures

Raises *QConnectionException*, *QWriterException*, *QReaderException*

async (*query*, **parameters*, ***options*)

Performs an asynchronous query and returns **without** retrieving of the response.

In typical use case, *query* is the name of the function to call and *parameters* are its parameters. When *parameters* list is empty, the query can be an arbitrary q expression (e.g. `0 +/ til 100`).

Calls a anonymous function with a single parameter:

```
>>> q.async('{til x}', 10)
```

Executes a q expression:

```
>>> q.async('til 10')
```

Parameters

- *query* (*string*) - query to be executed
- *parameters* (*list* or *None*) - parameters for the query

Options

- *single_char_strings* (*boolean*) - if `True` single char Python strings are encoded as q strings instead of chars, **Default**: `False`

Raises *QConnectionException*, *QWriterException*

receive (*data_only=True*, ***options*)

Reads and (optionally) parses the response from a q service.

Retrieves query result along with meta-information:

```
>>> q.query(qconnection.MessageType.SYNC,'{x}', 10)
>>> print(q.receive(data_only = False, raw = False))
QMessage: message type: 2, data size: 13, is_compressed: False, data: 10
```

Retrieves parsed query result:

```
>>> q.query(qconnection.MessageType.SYNC,'{x}', 10)
>>> print(q.receive(data_only = True, raw = False))
10
```

Retrieves not-parsed (raw) query result:

```
>>> from binascii import hexlify
>>> q.query(qconnection.MessageType.SYNC,'{x}', 10)
>>> print(hexlify(q.receive(data_only = True, raw = True)))
fa0a000000
```

Parameters

- *data_only* (*boolean*) - if `True` returns only data part of the message, otherwise returns data and message meta-information encapsulated in *QMessage* instance

Options

- *raw* (*boolean*) - if `True` returns raw data chunk instead of parsed data, **Default**: `False`
- *numpy_temporals* (*boolean*) - if `False` temporal vectors are backed by raw q representation (*QTemporalList*, *QTemporal*) instances, otherwise are represented as *numpy datetime64/timedelta64* arrays and atoms, **Default**: `False`

Returns depending on parameter flags: *QMessage* instance, parsed message, raw data

Raises *QReaderException*

qpython.qcollection module

class qpython.qcollection.**QList**(*spec=None*, *side_effect=None*, *return_value=sentinel.DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *parent=None*, *_spec_state=None*, *_new_name=''*, *_new_parent=None*, ***kwargs*)

Bases: `Mock`

An array object represents a q vector.

class qpython.qcollection.**QTemporalList**(*spec=None*, *side_effect=None*, *return_value=sentinel.DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *parent=None*, *_spec_state=None*, *_new_name=''*, *_new_parent=None*, ***kwargs*)

Bases: *qpython.qcollection.QList*

An array object represents a q vector of datetime objects.

raw(*idx*)

Gets the raw representation of the datetime object at the specified index.

```
>>> t = qlist(numpy.array([366, 121, qnull(QDATE)]), qtype=QDATE_LIST)
>>> print(t[0])
2001-01-01 [metadata(qtype=-14)]
>>> print(t.raw(0))
366
```

Parameters

- *idx* (*integer*) - array index of the datetime object to be retrieved

Returns raw representation of the datetime object

qpython.qcollection.**get_list_qtype**(*array*)

Finds out a corresponding qtype for a specified *QList/numpy.ndarray* instance.

Parameters

- *array* (*QList* or *numpy.ndarray*) - array to be checked

Returns *integer* - qtype matching the specified array object

qpython.qcollection.**qlist**(*array*, *adjust_dtype=True*, ***meta*)

Converts an input array to q vector and enriches object instance with meta data.

Returns a *QList* instance for non-datetime vectors. For datetime vectors *QTemporalList* is returned instead.

If parameter *adjust_dtype* is *True* and q type retrieved via *get_list_qtype()* doesn't match one provided as a *qtype* parameter guessed q type, underlying numpy.array is converted to correct data type.

qPython internally represents `(0x01;0x02;0xff)` q list as: `<class 'qpython.qcollection.QList'> dtype: int8 qtype: -4: [ 1 2 -1]`. This object can be created by calling the *qlist()* with following arguments:

•*byte numpy.array*:

```
>>> v = qlist(numpy.array([0x01, 0x02, 0xff], dtype=numpy.byte))
>>> print('%s dtype: %s qtype: %d: %s' % (type(v), v.dtype, v.meta.qtype, v))
<class 'qpython.qcollection.QList'> dtype: int8 qtype: -4: [ 1  2 -1]
```

•*int32 numpy.array* with explicit conversion to *QBYTE_LIST*:

```
>>> v = qlist(numpy.array([1, 2, -1]), qtype = QBYTE_LIST)
>>> print('%s dtype: %s qtype: %d: %s' % (type(v), v.dtype, v.meta.qtype, v))
<class 'qpython.qcollection.QList'> dtype: int8 qtype: -4: [ 1  2 -1]
```

•plain Python *integer* list with explicit conversion to *QBYTE_LIST*:

```
>>> v = qlist([1, 2, -1], qtype = QBYTE_LIST)
>>> print('%s dtype: %s qtype: %d: %s' % (type(v), v.dtype, v.meta.qtype, v))
<class 'qpython.qcollection.QList'> dtype: int8 qtype: -4: [ 1  2 -1]
```

•numpy datetime64 array with implicit conversion to *QDATE_LIST*:

```
>>> v = qlist(numpy.array([numpy.datetime64('2001-01-01'), numpy.datetime64(
↪'2000-05-01'), numpy.datetime64('NaT')], dtype='datetime64[D]'))
>>> print('%s dtype: %s qtype: %d: %s' % (type(v), v.dtype, v.meta.qtype, v))
<class 'qpython.qcollection.QList'> dtype: datetime64[D] qtype: -14: ['2001-
↪01-01' '2000-05-01' 'NaT']
```

•numpy datetime64 array with explicit conversion to *QDATE_LIST*:

```
>>> v = qlist(numpy.array([numpy.datetime64('2001-01-01'), numpy.datetime64(
↪'2000-05-01'), numpy.datetime64('NaT')], dtype='datetime64[D]'), qtype =␣
↪QDATE_LIST)
>>> print('%s dtype: %s qtype: %d: %s' % (type(v), v.dtype, v.meta.qtype, v))
<class 'qpython.qcollection.QList'> dtype: datetime64[D] qtype: -14: ['2001-
↪01-01' '2000-05-01' 'NaT']
```

Parameters

- *array* (*tuple*, *list*, *numpy.array*) - input array to be converted
- *adjust_dtype* (*boolean*) - determine whether data type of vector should be adjusted if it doesn't match default representation. **Default**: `True`

Note: numpy *datetime64* and *timedelta64* arrays are not converted to raw temporal vectors if *adjust_dtype* is `True`

Kwargs

- *qtype* (*integer* or *None*) - qtype indicator

Returns *QList* or *QTemporalList* - array representation of the list

Raises *ValueError*

class qpython.qcollection.**QDictionary**(*keys*, *values*)

Bases: `object`

Represents a q dictionary.

Dictionary examples:

```
>>> # q: 1 2!`abc`cdefgh
>>> print(QDictionary(qlist(numpy.array([1, 2], dtype=numpy.int64), qtype=QLONG_
↪LIST),
...                   qlist(numpy.array(['abc', 'cdefgh']), qtype = QSYMBOL_
↪LIST)))
[1 2]![`abc `cdefgh]
```

```
>>> # q: (1;2h;3.234;"4")!(`one;2 3;"456";(7;8 9))
>>> print(QDictionary([numpy.int64(1), numpy.int16(2), numpy.float64(3.234), '4'],
...                   [numpy.string_('one'), qlist(numpy.array([2, 3]),␣
↪qtype=QLONG_LIST), '456', [numpy.int64(7), qlist(numpy.array([8, 9]),␣
↪qtype=QLONG_LIST)]]))
[1, 2, 3.234, '4']![`one, QList([2, 3], dtype=int64), '456', [7, QList([8, 9],␣
↪dtype=int64)]]
```

Parameters

- *keys* (*QList*, *tuple* or *list*) - dictionary keys
- *values* (*QList*, *QTable*, *tuple* or *list*) - dictionary values

items()

Return a copy of the dictionary's list of `(key, value)` pairs.

iteritems()

Return an iterator over the dictionary's `(key, value)` pairs.

iterkeys()

Return an iterator over the dictionary's keys.

itervalues()

Return an iterator over the dictionary's values.

qpython.qcollection.**qtable**(*columns*, *data*, ***meta*)

Creates a QTable out of given column names and data, and initialises the meta data.

`QTable` is represented internally by *numpy.core.records.recarray*. Data for each column is converted to *QList* via *qlist()* function. If qtype indicator is defined for a column, this information is used for explicit array conversion.

Table examples:

```
>>> # q: flip `name`iq!(`Dent`Beeblebrox`Prefect;98 42 126)
>>> t = qtable(qlist(numpy.array(['name', 'iq']), qtype = QSYMBOL_LIST),
...      [qlist(numpy.array(['Dent', 'Beeblebrox', 'Prefect'])),
...       qlist(numpy.array([98, 42, 126], dtype=numpy.int64))])
>>> print('%s dtype: %s meta: %s: %s' % (type(t), t.dtype, t.meta, t))
<class 'qpython.qcollection.QTable'> dtype: [('name', 'S10'), ('iq', '<i8')]␣
↪meta: metadata(iq=-7, qtype=98, name=-11): [('Dent', 98L) ('Beeblebrox', 42L) (
↪'Prefect', 126L)]
```

```
>>> # q: flip `name`iq!(`Dent`Beeblebrox`Prefect;98 42 126)
>>> t = qtable(qlist(numpy.array(['name', 'iq']), qtype = QSYMBOL_LIST),
...            [qlist(['Dent', 'Beeblebrox', 'Prefect'], qtype = QSYMBOL_LIST),
...             qlist([98, 42, 126], qtype = QLONG_LIST)])
>>> print('%s dtype: %s meta: %s: %s' % (type(t), t.dtype, t.meta, t))
<class 'qpython.qcollection.QTable'> dtype: [('name', 'S10'), ('iq', '<i8')]␣
↪meta: metadata(iq=-7, qtype=98, name=-11): [('Dent', 98L) ('Beeblebrox', 42L) (
↪'Prefect', 126L)]
```

```
>>> # q: flip `name`iq!(`Dent`Beeblebrox`Prefect;98 42 126)
>>> t = qtable(['name', 'iq'],
...            [['Dent', 'Beeblebrox', 'Prefect'],
...             [98, 42, 126]],
...            name = QSYMBOL, iq = QLONG)
>>> print('%s dtype: %s meta: %s: %s' % (type(t), t.dtype, t.meta, t))
<class 'qpython.qcollection.QTable'> dtype: [('name', 'S10'), ('iq', '<i8')]␣
↪meta: metadata(iq=-7, qtype=98, name=-11): [('Dent', 98L) ('Beeblebrox', 42L) (
↪'Prefect', 126L)]
```

```
>>> # q: flip `name`iq`fullname!(`Dent`Beeblebrox`Prefect;98 42 126;("Arthur Dent
↪"; "Zaphod Beeblebrox"; "Ford Prefect"))
>>> t = qtable(('name', 'iq', 'fullname'),
...            [qlist(numpy.array(['Dent', 'Beeblebrox', 'Prefect']), qtype =␣
↪QSYMBOL_LIST),
...             qlist(numpy.array([98, 42, 126]), qtype = QLONG_LIST),
...             qlist(numpy.array(["Arthur Dent", "Zaphod Beeblebrox", "Ford␣
↪Prefect"]), qtype = QSTRING_LIST)])
<class 'qpython.qcollection.QTable'> dtype: [('name', 'S10'), ('iq', '<i8'), (
↪'fullname', 'O')] meta: metadata(iq=-7, fullname=0, qtype=98, name=-11): [('Dent
↪', 98L, 'Arthur Dent') ('Beeblebrox', 42L, 'Zaphod Beeblebrox') ('Prefect',␣
↪126L, 'Ford Prefect')]
```

Parameters

- *columns* (list of *strings*) - table column names
- *data* (list of lists) - list of columns containing table data

Kwargs

- *meta* (*integer*) - qtype for particular column

Returns *QTable* - representation of q table

Raises *ValueError*

class qpython.qcollection.**QKeyedTable**(*keys*, *values*)

Bases: object

Represents a q keyed table.

QKeyedTable is built with two QTables, one representing keys and the other values.

Keyed tables example:

```
>>> # q: ([eid:1001 1002 1003] pos:`d1`d2`d3;dates:(2001.01.01;2000.05.01;0Nd))
>>> t = QKeyedTable(qtable(['eid'],
...                 [qlist(numpy.array([1001, 1002, 1003]), qtype = QLONG_LIST)]),
...          qtable(['pos', 'dates'],
...                 [qlist(numpy.array(['d1', 'd2', 'd3']), qtype = QSYMBOL_LIST),
...                  qlist(numpy.array([366, 121, qnull(QDATE)]), qtype = QDATE_
↪LIST)]))
>>> print('%s: %s' % (type(t), t))
>>> print('%s dtype: %s meta: %s' % (type(t.keys), t.keys.dtype, t.keys.meta))
>>> print('%s dtype: %s meta: %s' % (type(t.values), t.values.dtype, t.values.
↪meta))
<class 'qpython.qcollection.QKeyedTable'>: [(1001L,) (1002L,) (1003L,)]![('d1',␣
↪366) ('d2', 121) ('d3', -2147483648)]
<class 'qpython.qcollection.QTable'> dtype: [('eid', '<i8')] meta:␣
↪metadata(qtype=98, eid=-7)
<class 'qpython.qcollection.QTable'> dtype: [('pos', 'S2'), ('dates', '<i4')]␣
↪meta: metadata(dates=-14, qtype=98, pos=-11)
```

Parameters

- *keys* (*QTable*) - table keys
- *values* (*QTable*) - table values

Raises *ValueError*

items()
Return a copy of the keyed table's list of `(key, value)` pairs.

iteritems()
Return an iterator over the keyed table's `(key, value)` pairs.

iterkeys()
Return an iterator over the keyed table's keys.

itervalues()
Return an iterator over the keyed table's values.

qpython.qtemporal module

class qpython.qtemporal.**QTemporal**(*dt*)
Bases: object

Represents a q temporal value.

The *QTemporal* wraps *numpy.datetime64* or *numpy.timedelta64* along with meta-information like qtype indicator.

Parameters

- *dt* (*numpy.datetime64* or *numpy.timedelta64*) - datetime to be wrapped

raw
Return wrapped datetime object.

Returns *numpy.datetime64* or *numpy.timedelta64* - wrapped datetime

qpython.qtemporal.**qtemporal**(*dt*, ***meta*)

Converts a *numpy.datetime64* or *numpy.timedelta64* to `QTemporal` and enriches object instance with given meta data.

Examples:

```
>>> qtemporal(numpy.datetime64('2001-01-01', 'D'), qtype=QDATE)
2001-01-01 [metadata(qtype=-14)]
>>> qtemporal(numpy.timedelta64(43499123, 'ms'), qtype=QTIME)
43499123 milliseconds [metadata(qtype=-19)]
>>> qtemporal(qnull(QDATETIME), qtype=QDATETIME)
nan [metadata(qtype=-15)]
```

Parameters

- *dt* (*numpy.datetime64* or *numpy.timedelta64*) - datetime to be wrapped

Kwargs

- *qtype* (*integer*) - qtype indicator

Returns *QTemporal* - wrapped datetime

qpython.qtemporal.**from_raw_qtemporal**(*raw*, *qtype*)

Converts raw numeric value to *numpy.datetime64* or *numpy.timedelta64* instance.

Actual conversion applied to raw numeric value depends on *qtype* parameter.

Parameters

- *raw* (*integer*, *float*) - raw representation to be converted
- *qtype* (*integer*) - qtype indicator

Returns *numpy.datetime64* or *numpy.timedelta64* - converted datetime

qpython.qtemporal.**to_raw_qtemporal**(*dt*, *qtype*)

Converts datetime/timedelta instance to raw numeric value.

Actual conversion applied to datetime/timedelta instance depends on *qtype* parameter.

Parameters

- *dt* (*numpy.datetime64* or *numpy.timedelta64*) - datetime/timedelta object to be converted
- *qtype* (*integer*) - qtype indicator

Returns *integer*, *float* - raw numeric value

qpython.qtemporal.**array_from_raw_qtemporal**(*raw*, *qtype*)

Converts *numpy.array* containing raw q representation to `datetime64`/`timedelta64` array.

Examples:

```
>>> raw = numpy.array([366, 121, qnull(QDATE)])
>>> print(array_from_raw_qtemporal(raw, qtype = QDATE))
['2001-01-01' '2000-05-01' 'NaT']
```

Parameters

- *raw* (*numpy.array*) - numpy raw array to be converted
- *qtype* (*integer*) - qtype indicator

Returns *numpy.array* - numpy array with `datetime64`/`timedelta64`

Raises *ValueError*

qpython.qtemporal.**array_to_raw_qtemporal**(*array*, *qtype*)

Converts *numpy.array* containing `datetime64`/`timedelta64` to raw q representation.

Examples:

```
>>> na_dt = numpy.arange('1999-01-01', '2005-12-31', dtype='datetime64[D]')
>>> print(array_to_raw_qtemporal(na_dt, qtype = QDATE_LIST))
[-365 -364 -363 ..., 2188 2189 2190]
>>> array_to_raw_qtemporal(numpy.arange(-20, 30, dtype='int32'), qtype = QDATE_
↪LIST)
Traceback (most recent call last):
  ...
ValueError: array.dtype is expected to be of type: datetime64 or timedelta64.␣
↪Was: int32
```

Parameters

- *array* (*numpy.array*) - numpy datetime/timedelta array to be converted
- *qtype* (*integer*) - qtype indicator

Returns *numpy.array* - numpy array with raw values

Raises *ValueError*

qpython.qtype module

The *qpython.qtype* module defines number of utility function which help to work with types mapping between q and Python.

This module declares supported q types as constants, which can be used along with conversion functions e.g.: *qcollection.qlist()* or *qtemporal.qtemporal()*.

List of q type codes:

q type name	q type code
QNULL	0x65
QGENERAL_LIST	0x00
QBOOL	-0x01
QBOOL_LIST	0x01
QGUID	-0x02
QGUID_LIST	0x02
QBYTE	-0x04
QBYTE_LIST	0x04
QSHORT	-0x05
QSHORT_LIST	0x05
QINT	-0x06
QINT_LIST	0x06
QLONG	-0x07
QLONG_LIST	0x07
QFLOAT	-0x08
QFLOAT_LIST	0x08
QDOUBLE	-0x09

Continued on next page

Table 6.1 – continued from previous page

q type name	q type code
QDOUBLE_LIST	0x09
QCHAR	-0x0a
QSTRING	0x0a
QSTRING_LIST	0x00
QSYMBOL	-0x0b
QSYMBOL_LIST	0x0b
QTIMESTAMP	-0x0c
QTIMESTAMP_LIST	0x0c
QMONTH	-0x0d
QMONTH_LIST	0x0d
QDATE	-0x0e
QDATE_LIST	0x0e
QDATETIME	-0x0f
QDATETIME_LIST	0x0f
QTIMESPAN	-0x10
QTIMESPAN_LIST	0x10
QMINUTE	-0x11
QMINUTE_LIST	0x11
QSECOND	-0x12
QSECOND_LIST	0x12
QTIME	-0x13
QTIME_LIST	0x13
QDICTIONARY	0x63
QKEYED_TABLE	0x63
QTABLE	0x62
QLAMBDA	0x64
QUNARY_FUNC	0x65
QBINARY_FUNC	0x66
QTERNARY_FUNC	0x67
QCOMPOSITION_FUNC	0x69
QADVERB_FUNC_106	0x6a
QADVERB_FUNC_107	0x6b
QADVERB_FUNC_108	0x6c
QADVERB_FUNC_109	0x6d
QADVERB_FUNC_110	0x6e
QADVERB_FUNC_111	0x6f
QPROJECTION	0x68
QERROR	-0x80

qpython.qtype.**qnull**(*qtype*)

Retrieve null value for requested q type.

Parameters

- *qtype* (*integer*) - qtype indicator

Returns null value for specified q type

qpython.qtype.**is_null**(*value*, *qtype*)

Checks whether given value matches null value for a particular q type.

Parameters

- *qtype* (*integer*) - qtype indicator

Returns *boolean* - `True` if value is considered null for given type `False` otherwise

exception `qpython.qtype.`**`QException`**
Bases: `exceptions.Exception`

Represents a q error.

class `qpython.qtype.`**`QFunction`**(*qtype*)
Bases: `object`

Represents a q function.

class `qpython.qtype.`**`QLambda`**(*expression*)
Bases: *qpython.qtype.QFunction*

Represents a q lambda expression.

Note: *expression* is trimmed and required to be valid q function (`{..}`) or k function (`k){..}`).

Parameters

- *expression* (*string*) - lambda expression

Raises *ValueError*

class `qpython.qtype.`**`QProjection`**(*parameters*)
Bases: *qpython.qtype.QFunction*

Represents a q projection.

Parameters

- *parameters* (*list*) - list of parameters for lambda expression

class `qpython.qtype.`**`Mapper`**(*call_map*)
Bases: `object`

Utility class for creating function execution map via decorators.

Parameters

- *call_map* (*dictionary*) - target execution map

qpython.qreader module

exception `qpython.qreader.`**`QReaderException`**
Bases: `exceptions.Exception`

Indicates an error raised during data deserialization.

class `qpython.qreader.`**`QMessage`**(*data*, *message_type*, *message_size*, *is_compressed*)
Bases: `object`

Represents a single message parsed from q protocol. Encapsulates data, message size, type, compression flag.

Parameters

- *data* - data payload
- *message_type* (one of the constants defined in *MessageType*) - type of the message

- *message_size* (*integer*) - size of the message
- *is_compressed* (*boolean*) - indicates whether message is compressed

data
Parsed data.

type
Type of the message.

is_compressed
Indicates whether source message was compressed.

size
Size of the source message.

class qpython.qreader.**QReader**(*stream*, *encoding='latin-1'*)
Bases: object

Provides deserialization from q IPC protocol.

Parameters

- *stream* (*file object* or *None*) - data input stream
- *encoding* (*string*) - encoding for characters parsing

Attrbutes

- *_reader_map* - stores mapping between q types and functions responsible for parsing into Python objects

read(*source=None*, ***options*)
Reads and optionally parses a single message.

Parameters

- *source* - optional data buffer to be read, if not specified data is read from the wrapped stream

Options

- *raw* (*boolean*) - indicates whether read data should parsed or returned in raw byte form
- *numpy_temporals* (*boolean*) - if `False` temporal vectors are backed by raw q representation (*QTemporalList*, *QTemporal*) instances, otherwise are represented as *numpy datetime64*/*timedelta64* arrays and atoms, **Default**: `False`

Returns *QMessage* - read data (parsed or raw byte form) along with meta information

read_header(*source=None*)
Reads and parses message header.

Note: `read_header()` wraps data for further reading in internal buffer

Parameters

- *source* - optional data buffer to be read, if not specified data is read from the wrapped stream

Returns *QMessage* - read meta information

read_data(*message_size*, *is_compressed=False*, ***options*)
Reads and optionally parses data part of a message.

Note: `read_header()` is required to be called before executing the `read_data()`

Parameters

- *message_size* (*integer*) - size of the message to be read
- *is_compressed* (*boolean*) - indicates whether data is compressed

Options

- *raw* (*boolean*) - indicates whether read data should parsed or returned in raw byte form
- *numpy_temporals* (*boolean*) - if `False` temporal vectors are backed by raw q representation (*QTemporalList*, *QTemporal*) instances, otherwise are represented as *numpy datetime64/timedelta64* arrays and atoms, **Default**: `False`

Returns read data (parsed or raw byte form)

class **BytesBuffer**
Bases: `object`

Utility class for reading bytes from wrapped buffer.

endianness
Gets the endianness.

wrap(*data*)
Wraps the data in the buffer.
Parameters
- *data* - data to be wrapped

skip(*offset=1*)
Skips reading of *offset* bytes.
Parameters
- *offset* (*integer*) - number of bytes to be skipped

raw(*offset*)
Gets *offset* number of raw bytes.
Parameters
- *offset* (*integer*) - number of bytes to be retrieved

Returns raw bytes

get(*fmt*, *offset=None*)
Gets bytes from the buffer according to specified format or *offset*.
Parameters
- *fmt* (struct format) - conversion to be applied for reading
- *offset* (*integer*) - number of bytes to be retrieved

Returns unpacked bytes

get_byte()
Gets a single byte from the buffer.
Returns single byte

get_int()
Gets a single 32-bit integer from the buffer.
Returns single integer

get_symbol()
Gets a single, \x00 terminated string from the buffer.
Returns \x00 terminated string

get_symbols(*count*)
Gets count \x00 terminated strings from the buffer.
Parameters
- *count* (*integer*) - number of strings to be read

Returns list of \x00 terminated string read from the buffer

qpython.qwriter module

exception qpython.qwriter.**QWriterException**
Bases: exceptions.Exception

Indicates an error raised during data serialization.

class qpython.qwriter.**QWriter**(*stream*, *protocol_version*, *encoding='latin-1'*)
Bases: object

Provides serialization to q IPC protocol.

Parameters
- *stream* (*socket* or *None*) - stream for data serialization
- *protocol_version* (*integer*) - version IPC protocol
- *encoding* (*string*) - encoding for characters serialization

Attrbutes
- *_writer_map* - stores mapping between Python types and functions responsible for serializing into IPC representation

write(*data*, *msg_type*, ***options*)
Serializes and pushes single data object to a wrapped stream.

Parameters
- *data* - data to be serialized
- *msg_type* (one of the constants defined in *MessageType*) - type of the message

Options
- *single_char_strings* (*boolean*) - if True single char Python strings are encoded as q strings instead of chars, **Default**: False

Returns if wraped stream is None serialized data, otherwise None

CHAPTER 7

Indices and tables

- genindex
- search

Python Module Index

q

Index

R

S

T

W