PyQ Documentation

Release 4.0.1

Enlightenment Research, LLC.

Mar 15, 2017

Table of Contents

PyQ brings the Python programming language to the kdb+ database. It allows developers to seamlessly integrate Python and q codes in one application. This is achieved by bringing the Python and q interpreters in the same process so that codes written in either of the languages operate on the same data. In PyQ, Python and q objects live in the same memory space and share the same data.

CHAPTER 1

Quick start

Install pyq

Don't have pyq installed? Run

```
$ pip install \
  -i https://pyq.enlnt.com \
  --no-binary pyq pyq
```

First, make sure that PyQ is *installed* and *up-to-date*. Start an interactive session:

```
$ pyq
```

Import the `q` object from `pyq` and the `date` class from the standard library module `datetime`:

```
>>> from pyq import q
>>> from datetime import date
```

Drop to the *q) prompt* and create an empty `trade` table:

```
>>> q()
q)trade:([]date:();sym:();qty:())
```

Get back to the Python prompt and insert some data into the `trade` table:

```
q)\
>>> q.insert('trade', (date(2006,10,6), 'IBM', 200))
k(',0')
>>> q.insert('trade', (date(2006,10,6), 'MSFT', 100))
k(',1')
```

(In the following we will skip `q()` and `\` commands that switch between `q` and Python.)

Display the result:

```
>>> q.trade.show()
date       sym  qty
-------------------
2006.10.06 IBM  200
2006.10.06 MSFT 100
```

Define a function in `q`:

```
q)f:{[s;d]select from trade where sym=s,date=d}
```

Call the `q` function from python and pretty-print the result:

```
>>> x = f('IBM', date(2006,10,6))
>>> x.show()
date       sym qty
------------------
2006.10.06 IBM 200
```

For an enhanced interactive shell, use `pyq` to start IPython:

```
$ pyq -m IPython
```

See the *ipython section* for details.

What's New in PyQ 4.0

Release 4.0.1

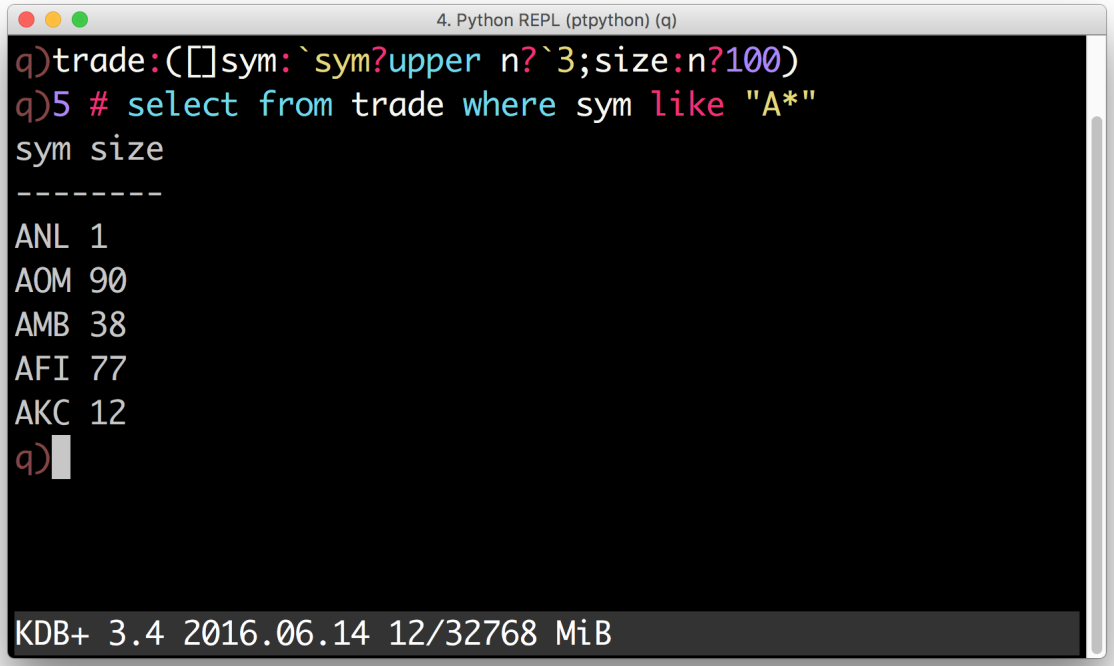
Date Mar 15, 2017

Summary – Release highlights

- Enhanced q) prompt with syntax highlighting.
- New operators: <<, >> and `@`.
- Improved means for constructing `K` objects of arbitrary types.
- Type casts using attribute syntax.
- Improved numpy interoperability.
- Restored support for KDB+ 2.x.
- Better documentation.
- More `k.h` functions are exposed to Python internally.
- Added convenience scripts for starting different interactive sessions.
- Additional conversions between `K` and native Python objects.
- Redesigned adverbs

Enhanced `q)` prompt

The `q)` prompt will now use the prompt toolkit when available to provide a separate command history, q syntax highlighting and a status bar displaying system information.



New operators

Three new operators are defined for `K` objects: `<<`, `>>` and `@`.

Shift operators

Shift operators `<<` and `>>` can now be used to shift elements in `K` lists:

```
>>> q.til(10) << 3
k('3 4 5 6 7 8 9 0N 0N 0N')
>>> q.til(10) >> 3
k('0N 0N 0N 0 1 2 3 4 5 6')
```

The `@` operator

Users of Python 3.5 or later can now use the new binary operator `@` to call q functions without using parentheses:

```
>>> q.til @ 5
k('0 1 2 3 4')
```

The same operator between two functions creates a function composition. For example, the dot product can be defined succinctly as

```
>>> dot = q.sum @ q('*')
>>> dot([1, 2, 3], [3, 2, 1])
k('10')
```

Typed constructors and casts

Atoms and lists of like atoms can now be constructed from Python objects using typed constructors. For example, by default, a list of strings passed to the default `K` constructor becomes a symbol list:

```
>>> colors = K(['white', 'blue', 'red'])
>>> colors
k('`white`blue`red')
```

If you want to create a list of strings, you can use a typed constructor:

```
>>> K.string(["Donald E. Knuth", "Edsger W. Dijkstra"])
k('("Donald E. Knuth";"Edsger W. Dijkstra")')
```

If you already have a symbol list and want to convert it to strings, you can use the attribute access notation to perform the cast:

```
>>> colors.string
k('("white";"blue";"red")')
```

Similar operations can be performed with numeric data. For example, to create a matrix of single-precision floats (real), call

```
>>> m = K.real([[1, 0, 0],
...             [0, 1, 0],
...             [0, 0, 1]])
>>> m
k('(1 0 0e;0 1 0e;0 0 1e)')
```

To cast the result to booleans — access the `boolean` attribute:

```
>>> m.boolean.show()
100b
010b
001b
```

Unlike q, Python does not have special syntax for missing values and infinities. Those values can now be created in PyQ by accessing `na` and `inf` attributes on the typed constructors:

```
>>> for x in [K.int, K.float, K.date, K.timespan]:
...     print(x.na, x.inf)
0Ni 0Wi
0n 0w
0Nd 0Wd
0Nn 0Wn
```

Interoperability with NumPy

Matrices and arrays of higher dimensions

Arrays with `ndim > 1` can now be passed to `q` and they become nested lists. For example:

```
>>> q.x = numpy.arange(12, dtype=float).reshape((2, 3, 2))
>>> q.x
k('((0 1f;2 3f;4 5f);(6 7f;8 9f;10 11f))')
```

Similarly, `ndim > 1` arrays can be constructed from lists of regular shape:

```
>>> numpy.array(q.x)
array([[[  0.,   1.],
        [  2.,   3.],
        [  4.,   5.]],

       [[  6.,   7.],
        [  8.,   9.],
        [ 10.,  11.]]])
```

Times, dates and timedeltas

Prior to 4.0, conversion of temporal data to NumPy arrays would expose internal integer values. For example, a list of months

```
>>> months = q('2001.01m + til 3')
```

would become an integer array when converted to NumPy:

```
>>> numpy.array(months).tolist()
[12, 13, 14]
```

Now, an array of type datetime64 is returned:

```
>>> numpy.array(months)
array(['2001-01', '2001-02', '2001-03'], dtype='datetime64[M]')
```

Note that the resulting array has different numeric values and cannot share the data with the `K` object. To share the data and/or to get an array as in older versions, one should use the new `data` attribute:

```
>>> a = numpy.asarray(months.data)
>>> a.tolist()
[12, 13, 14]
```

An array constructed from the `data` attribute will use the same underlying storage. This means that changing the array will change the `K` object.

```
>>> a[:] += 998*12
>>> months
k('2999.01 2999.02 2999.03m')
```

Additional conversions

Complex numbers

Complex numbers can now be passed to and obtained from kdb+. When passed to kdb+, complex numbers are automatically converted to dictionaries with keys "re" and "im" and lists of complex numbers are converted to tables with columns "re" and "im".

```
>>> q.z = [1 + 2j, 3 + 4j, 5 + 6j]
>>> q.z.show()
re im
-----
1  2
3  4
5  6
>>> [complex(x) for x in q.z]
[(1+2j), (3+4j), (5+6j)]
```

Path objects

`Path` objects can now be used where q path handle symbols are expected

```
>>> import pathlib
>>> path = pathlib.Path('xyz')
>>> q.set(path, 42)
k(``:xyz')
>>> q.get(path)
k('42')
```

Named tuples

Named tuples are now converted to dictionaries:

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', 'x,y')
>>> q.point = Point(1, 2)
>>> q.point
k(``x`y!1 2')
```

As a consequence, a uniform list of named tuples is converted to a table:

```
>>> q.points = [Point(1, 2), Point(3, 4), Point(5, 6)]
>>> q.points.show()
x y
---
1 2
3 4
5 6
```

Redesigned adverbs

Adverbs can now be used on functions with different ranks. For example, `scan` and `over` can be used with monadic functions. To illustrate, the following code generates a Pascal triangle:

```
>>> f = q('{(0,x)+x,0}')
>>> f.scan(6, 1).show()
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

If only the last row is of interest – use `over`:

```
>>> f.over(6, 1)
k('1 6 15 20 15 6 1')
```

Installation

PyQ can be installed using the standard Python package management tool - pip. See Installing Python Modules for details. To install the latest version, run the following command

```
$ pip install -i https://pyq.enlnt.com --no-binary pyq pyq
```

Prerequisites

OS Support

PyQ has been tested and is supported on Linux and macOS 10.11 or later.

PyQ has support for Solaris, but has not been tested recently.

Windows is not supported yet.

Required Software

- kdb+ 2.8 or later;
- Python 2.7, or 3.5 or later;
- GNU make, gcc or clang.

Installing from the package repository

Use following pip command to install the latest version of PyQ into your environment.

```
$ pip install -i https://pyq.enlnt.com --no-binary pyq pyq
```

To install another version, specify which version you would like to install:

```
$ pip install -i https://pyq.enlnt.com --no-binary pyq pyq==3.8
```

Installing from source code

1. Get the source code using one of the following:
   - Clone the Github repository

     ```
     $ git clone https://github.com/enlnt/pyq.git
     ```

   - Download the source archive as a tar file or a zip file and extract it.
2. Install the sources into your environment using pip:

   ```
   $ pip install <path to the source>
   ```

Installing PyQ into a virtual environment

PyQ was designed to work inside virtual environments. You can setup your system to use different versions of Python and/or kdb+ by using separate virtual environments.

In order to create a virtual environment, you need to install the virtualenv package:

```
$ [sudo] pip install virtualenv
```

Create a new virtualenv and activate it:

```
$ virtualenv path/to/virtualenv
$ source path/to/virtualenv/bin/activate
```

Download kdb+ and save into your `~/Downloads` folder. Extract it into virtualenv:

```
$ unzip ${HOME}/Downloads/macosx.zip -d ${VIRTUAL_ENV}
```

If you have licensed version of the kdb+, you should create directory for it first:

```
$ mkdir -p ${VIRTUAL_ENV}/q && unzip path/to/m64.zip -d ${VIRTUAL_ENV}/q
```

Copy your kdb+ license file to `${VIRTUAL_ENV}/q` or set the `QLIC` environment variable to the directory containing the license file and add it to the virtualenv's `activate` file:

```
$ echo "export QLIC=path/to/qlic" >> ${VIRTUAL_ENV}/bin/activate
$ source ${VIRTUAL_ENV}/bin/activate
```

Install PyQ:

```
$ pip install -i https://pyq.enlnt.com --no-binary pyq pyq
```

Keeping PyQ up-to-date

You can upgrade PyQ to the latest version by running:

```
pip install -i https://pyq.enlnt.com --no-binary pyq -U pyq
```

Installing 32-bit PyQ with the free 32-bit kdb+ and Python 3.6 on 64-bit CentOS 7

Note: This guide was designed for installing Python 3.6. If you're looking to use Python 2.7, you can follow this guide by replacing `3.6.0` with `2.7.13` where necessary.

1. Install development tools and libraries required to build 32-bit Python

```
$ sudo yum install gcc gcc-c++ rpm-build subversion git zip unzip bzip2 \
  libgcc.i686 glibc-devel.i686 glibc.i686 zlib-devel.i686 \
  readline-devel.i686 gdbm-devel.i686 openssl-devel.i686 ncurses-devel.i686 \
  tcl-devel.i686 libdb-devel.i686 bzip2-devel.i686 sqlite-devel.i686 \
  tk-devel.i686 libpcap-devel.i686 xz-devel.i686 libffi-devel.i686
```

2. Download, compile and install the 32-bit version of Python 3.6.0

We are going to install Python 3.6 into `/opt/python3.6.i686`:

```
$ mkdir -p ${HOME}/Archive ${HOME}/Build
$ sudo mkdir -p /opt/python3.6.i686
$ curl -Ls http://www.python.org/ftp/python/3.6.0/Python-3.6.0.tgz \
  -o ${HOME}/Archive/Python-3.6.0.tgz
$ tar xzvf ${HOME}/Archive/Python-3.6.0.tgz -C ${HOME}/Build
$ cd ${HOME}/Build/Python-3.6.0
$ export CFLAGS=-m32 LDFLAGS=-m32
$ ./configure --prefix=/opt/python3.6.i686 --enable-shared
$ LD_RUN_PATH=/opt/python3.6.i686/lib make
$ sudo make install
$ unset CFLAGS LDFLAGS
```

Let's confirm we've got 32-bit Python on our 64-bit system:

```
$ uname -mip
x86_64 x86_64 x86_64
$ /opt/python3.6.i686/bin/python3.6 \
  -c "import platform; print(platform.processor(), platform.architecture())"
x86_64 ('32bit', 'ELF')
```

Yes, it is exactly what we desired.

3. Install virtualenv into Python installation

We are going to use virtual environments, download and extract virtualenv package:

```
$ curl -Ls https://pypi.org/packages/source/v/virtualenv/virtualenv-15.1.0.tar.gz \
  -o ${HOME}/Archive/virtualenv-15.1.0.tar.gz
$ tar xzf ${HOME}/Archive/virtualenv-15.1.0.tar.gz -C ${HOME}/Build
```

4. Create 32-bit Python virtual environment

Create a virtual environment:

```
$ /opt/python3.6.i686/bin/python3.6 ${HOME}/Build/virtualenv-15.1.0/virtualenv.py \
  ${HOME}/Work/pyq3
```

Enter the virtual environment we've just created, confirm we've got 32-bit Python in it:

```
(pyq3) $ source ${HOME}/Work/pyq3/bin/activate
(pyq3) $ python -c "import struct; print(struct.calcsize('P') * 8)"
32
```

5. Download the 32-bit Linux x86 version of kdb+ from kx.com

Download kdb+ by following this link.

Save downloaded file as *${HOME}/Work/linux-x86.zip*.

6. Extract kdb+ and install PyQ

Extract downloaded file:

```
(pyq3) $ unzip ${HOME}/Work/linux-x86.zip -d ${VIRTUAL_ENV}
```

Install PyQ (note, PyQ 3.8.2 or newer required):

```
(pyq3) $ pip install -i https://pyq.enlnt.com --no-binary pyq pyq>=3.8.2
```

6. Use PyQ

Start PyQ:

```
(pyq3) $ pyq
```

```
>>> import platform
>>> platform.processor()
'x86_64'
>>> platform.architecture()
('32bit', 'ELF')
>>> from pyq import q
>>> q.til(10)
k('0 1 2 3 4 5 6 7 8 9')
```

Python for kdb+

Introduction

Kdb+, a high-performance database system comes with a programming language (q) that may be unfamiliar to many programmers. PyQ lets you enjoy the power of kdb+ in a comfortable environment provided by a mainstream programming language. In this guide we will assume that the reader has a working knowledge of Python, but we will explain the q language concepts as we encounter them.

The q namespace

Meet `q` - your portal to kdb+. Once you import *q* from `pyq`, you get access to over 170 functions:

```
>>> from pyq import q
>>> dir(q)
['abs', 'acos', 'aj', 'aj0', 'all', 'and_', 'any', 'asc', 'asin', ...]
```

These functions should be familiar to anyone who knows the q language and this is exactly what these functions are: q functions repackaged so that they can be called from Python. Some of the q functions are similar to Python builtins or `math` functions which is not surprising because q like Python is a complete general purpose language. In the following sections we will systematically draw an analogy between q and Python functions and explain the differences between them.

The til function

Since Python does not have a language constructs to loop over integers, many Python tutorials introduce the `range()` function early on. In the q language, the situation is similar and the function that produces a sequence of integers is called "til". Mnemonically, `q.til(n)` means "Count from zero 'til *n*":

```
>>> q.til(10)
k('0 1 2 3 4 5 6 7 8 9')
```

The return value of a q function is always an instance of the class `K` which will be described in the next chapter. In the case of `q.til(n)`, the result is a `K` vector which is similar to Python list. In fact, you can get the Python list by simply calling the `list()` constructor on the q vector:

```
>>> list(_)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

While useful for illustrative purposes, you should avoid converting `K` vectors to Python lists in real programs. It is often more efficient to manipulate `K` objects directly. For example, unlike `range()`, *`til()`* does not have optional start or step arguments. This is not necessary because you can do arithmetic on the `K` vectors to achieve a similar result:

```
>>> range(10, 20, 2) == 10 + 2 * q.til(5)
True
```

Many q functions are designed to "map" themselves automatically over sequences passed as arguments. Those functions are called "atomic" and will be covered in the next section. The *`til()`* function is not atomic, but it can be mapped explicitly:

```
>>> q.til.each(range(5)).show()
`long$()
,0
0 1
0 1 2
0 1 2 3
```

The last example requires some explanation. First we have used the *`show()`* method to provide a nice multi-line display of a list of vectors. This method is available for all `K` objects. Second, the first line in the display shows and empty list of type "long". Note that unlike Python lists `K` vectors come in different types and *`til()`* returns vectors of type "long". Finally, the second line in the display starts with "," to emphasize that this is a vector of size 1 rather than an atom.

The *`each()`* adverb is similar to Python's `map()`, but is often much faster.

```
>>> q.til.each(range(5)) == map(q.til, range(5))
True
```

Atomic functions

As we mentioned in the previous section, atomic functions operate on numbers or lists of numbers. When given a number, an atomic function acts similarly to its Python analogue.

Compare

```
>>> q.exp(1)
k('2.718282')
```

and

```
>>> math.exp(1)
2.718281828459045
```

Note: Want to see more digits? Set `q` display precision using the *`system()`* function:

```
>>> q.system(b"P 16")
k('::')
>>> q.exp(1)
k('2.718281828459045')
```

Unlike their native Python analogues, atomic `q` functions can operate on sequences:

```
>>> q.exp(range(5))
k('1 2.718282 7.389056 20.08554 54.59815')
```

The result in this case is a `K` vector whose elements are obtained by applying the function to each element of the given sequence.

Mathematical functions

As you can see in the table below, most of the mathematical functions provided by q are similar to the Python standard library functions in the `math` module.

Table 1.1: Mathematical functions

q	Python	Return
neg()	operator.neg()	the negative of the argument
abs()	abs()	the absolute value
signum()		±1 or 0 depending on the sign of the argument
sqrt()	math.sqrt()	the square root of the argument
exp()	math.exp()	e raised to the power of the argument
log()	math.log()	the natural logarithm (base e) of the argument
cos()	math.cos()	the cosine of the argument
sin()	math.sin()	the sine of the argument
tan()	math.tan()	the tangent of the argument
acos()	math.acos()	the arc cosine of the argument
asin()	math.asin()	the arc sine of the argument
atan()	math.atan()	the arc tangent of the argument
ceiling()	math.ceil()	the smallest integer >= the argument
floor()	math.floor()	the largest integer <= the argument
reciprocal()		1 divided by the argument

Other than being able to operate on lists of of numbers, q functions differ from Python functions in a way they treat out of domain errors.

Where Python functions raise an exception,

```
>>> math.log(0)
Traceback (most recent call last):
  ...
ValueError: math domain error
```

q functions return special values:

```
>>> q.log([-1, 0, 1])
k('0n -0w 0')
```

The null function

Unlike Python, q allows division by zero. The reciprocal of zero is infinity that shows up as 0w or 0W in displays.

```
>>> q.reciprocal(0)
k('0w')
```

Multiplying infinity by zero produces a null value that generally indicates missing data

```
>>> q.reciprocal(0) * 0
k('0n')
```

Null values and infinities can also appear as a result of applying a mathematical function to numbers outside of its domain:

```
>>> q.log([-1, 0, 1])
k('0n -0w 0')
```

The *null()* function returns 1b (boolean true) when given a null value and 0b otherwise. For example, wen applied to the output of the *log()* function from the previous example, it returns

```
>>> q.null(_)
k('100b')
```

Aggregation functions

Aggregation functions (also known as reduction functions) are functions that given a sequence of atoms produce an atom. For example,

```
>>> sum(range(10))
45
>>> q.sum(range(10))
k('45')
```

Table 1.2: Aggregation functions

q	Python	Return
sum()	sum()	the sum of the elements
prd()		the product of the elements
all()	all()	1b if all elements are nonzero, 0b otherwise
any()	any()	1b if any of the elements is nonzero, 0b otherwise
min()	min()	the smallest element
max()	max()	the largest element
avg()	statistics.mean()	the arithmetic mean
var()	statistics.pvariance()	the population variance
dev()	statistics.pstdev()	the square root of the population variance
svar()	statistics.variance()	the sample variance
sdev()	statistics.stdev()	the square root of the sample variance

Accumulation functions

Given a sequence of numbers, one may want to compute not just total sum, but all the intermediate sums as well. In q, this can be achieved by applying the sums function to the sequence:

```
>>> q.sums(range(10))
k('0 1 3 6 10 15 21 28 36 45')
```

Table 1.3: Accumulation functions

q	Return
pyq.q.sums()	the cumulative sums of the elements
pyq.q.prds()	the cumulative products of the elements
pyq.q.maxs()	the maximums of the prefixes of the argument
pyq.q.mins()	the minimums of the prefixes of the argument

There are no direct analogues of these functions in the Python standard library, but the itertools.accumulate() function provides similar functionality:

```
>>> list(itertools.accumulate(range(10)))
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

Passing operator.mul(), max() or min() as the second optional argument to itertools.accumulate(), one can get analogues of *pyq.q.prds()*, *pyq.q.maxs()* and *pyq.q.mins()*.

Sliding window statistics

- *mavg()*
- *mcount()*
- *mdev()*
- *mmax()*
- *mmin()*
- *msum()*

Uniform functions

Uniform functions are functions that take a list and return another list of the same size.

- *reverse()*
- *ratios()*
- *deltas()*
- *differ()*
- *next()*
- *prev()*
- *fills()*

Set operations

- *except_()*
- *inter()*
- *union()*

Sorting and searching

Functions *asc()* and *desc()* sort lists in ascending and descending order respectively:

```
>>> a = [9, 5, 7, 3, 1]
>>> q.asc(a)
k('`s#1 3 5 7 9')
>>> q.desc(a)
k('9 7 5 3 1')
```

Note: The `s# prefix that appears in the display of the output for the *asc()* function indicates that the resulting vector has a sorted attribute set. An attribute can be queried by calling the *attr()* function or accessing the *attr* property of the result:

```
>>> s = q.asc(a)
>>> q.attr(s)
k('`s')
>>> s.attr
k('`s')
```

When the *asc()* function gets a vector with the `s` attribute set, it skips sorting and immediately returns the same vector.

Functions *iasc()* and *idesc()* return the indices indicating the order in which the elements of the incoming list should be taken to make them sorted:

```
>>> q.iasc(a)
k('4 3 1 2 0')
```

Sorted lists can be efficiently searched using *bin()* and *binr()* functions. As the names suggest, both use binary search to locate the position the element that is equal to the search key, but in the case when there is more than one such element, *binr()* returns the index of the first match while *bin()* returns the index of the last.

```
>>> q.binr([10, 20, 20, 20, 30], 20)
k('1')
>>> q.bin([10, 20, 20, 20, 30], 20)
k('3')
```

When no matching element can be found, *binr()* (*bin()*) returns the index of the position before (after) which the key can be inserted so that the list remains sorted.

```
>>> q.binr([10, 20, 20, 20, 30], [5, 15, 20, 25, 35])
k('0 1 1 4 5')
>>> q.bin([10, 20, 20, 20, 30], [5, 15, 20, 25, 35])
k('-1 0 3 3 4')
```

In the Python standard library similar functionality is provided by the `bisect` module.

```
>>> [bisect.bisect_left([10, 20, 20, 20, 30], key) for key in [5, 15, 20, 25, 35]]
[0, 1, 1, 4, 5]
>>> [-1 + bisect.bisect_right([10, 20, 20, 20, 30], key) for key in [5, 15, 20, 25,␣
↪35]]
[-1, 0, 3, 3, 4]
```

Note that while *binr()* and `bisect.bisect_left()` return the same values, *bin()* and `bisect.bisect_right()` are off by 1.

Q does not have a named function for searching in an unsorted list because it uses the `?` operator for that. We can easily expose this functionality in PyQ as follows:

```
>>> index = q('?')
>>> index([10, 30, 20, 40], [20, 25])
k('2 4')
```

Note that our home-brew `index` function is similar to the `list.index()` method, but it returns the one after last index when the key is not found while `list.index()` raises an exception.

```
>>> list.index([10, 30, 20, 40], 20)
2
>>> list.index([10, 30, 20, 40], 25)
Traceback (most recent call last):
  ...
ValueError: 25 is not in list
```

If you are not interested in the index, but only want to know whether the keys can be found in a list, you can use the *in_()* function:

```
>>> q.in_([20, 25], [10, 30, 20, 40])
k('10b')
```

Note: The *q.in_* function has a trailing underscore because otherwise it would conflict with the Python `in`.

From Python to kdb+

You can pass data from Python to kdb+ by assigning to `q` attributes. For example,

```
>>> q.i = 42
>>> q.a = [1, 2, 3]
>>> q.t = ('Python', 3.5)
>>> q.d = {'date': date(2012, 12, 12)}
>>> q.value.each(['i', 'a', 't', 'd']).show()
42
1 2 3
(`Python;3.5)
(,`date)!,2012.12.12
```

Note that Python objects are automatically converted to kdb+ form when they are assigned in the `q` namespace, but when they are retrieved, Python gets a "handle" to kdb+ data.

For example, passing an `int` to `q` results in

```
>>> q.i
k('42')
```

If you want a Python integer instead, you have to convert explicitly

```
>>> int(q.i)
42
```

This will be covered in more detail in the next section.

You can also create kdb+ objects by calling `q` functions that are also accessible as `q` attributes. For example,

```
>>> q.til(5)
k('0 1 2 3 4')
```

Some q functions don't have names because q uses special characters. For example, to generate random data in q you should use the `?` function (operator). While PyQ does not supply a Python name for `?`, you can easily add it to your own toolkit:

```
>>> rand = q('?')
```

And use it as you would any other Python function

```
>>> x = rand(10, 2)  # generates 10 random 0's or 1's (coin toss)
```

From kdb+ to Python

In many cases your data is already stored in kdb+ and PyQ philosophy is that it should stay there. Rather than converting kdb+ objects to Python, manipulating Python objects and converting them back to kdb+, PyQ lets you work directly with kdb+ data as if it was already in Python.

For example, let us retrieve the release date from kdb+:

```
>>> d1 = q('.z.k')
```

add 30 days to get another date

```
>>> d2 = d1 + 30
```

and find the difference in whole weeks

```
>>> (d2 - d1) % 7
k('2')
```

Note that the result of operations are (handles to) kdb+ objects. The only exceptions to this rule are indexing and iteration over simple kdb+ vectors. These operations produce Python scalars

```
>>> list(q.a)
[1, 2, 3]
>>> q.a[-1]
3
```

In addition to Python operators, one invoke q functions on kdb+ objects directly from Python using convenient attribute access / method call syntax.

For example

```
>>> q.i.neg.exp.log.mod(5)
k('3f')
```

Note that the above is equivalent to

```
>>> q.mod(q.log(q.exp(q.neg(q.i))), 5)
k('3f')
```

but shorter and closer to `q` syntax

```
>>> q('(log exp neg i)mod 5')
k('3f')
```

The difference being that in q, functions are applied right to left, by in PyQ left to right.

Finally, if q does not provide the function that you need, you can unleash the full power of numpy or scipy on your kdb+ data.

```
>>> numpy.log2(q.a)
array([ 0.        ,  1.        ,  1.5849625])
```

Note that the result is a numpy array, but you can redirect the output back to kdb+. To illustrate this, create a vector of 0s in kdb+

```
>>> b = q.a * 0.0
```

and call a numpy function on one kdb+ object redirecting the output to another:

```
>>> numpy.log2(q.a, out=numpy.asarray(b))
```

The result of a numpy function is now in the kdb+ object

```
>>> b
k('0 1 1.584963')
```

Working with files

Kdb+ uses unmodified host file system to store data and therefore q has excellent support for working with files. Recall that we can send Python objects to kdb+ by simply assigning them to a `q` attribute:

```
>>> q.data = range(10)
```

This code saves 10 integers in kdb+ memory and makes a global variable `data` available to kdb+ clients, but it does not save the data in any persistent storage. To save `data` is a file "data", we can simply call the `pyq.q.save` function as follows:

```
>>> q.save('data')
k('`:data')
```

Note that the return value of the `pyq.q.save` function is a `K` symbol that is formed by prepending ':' to the file name. Such symbols are known as file handles in q. Given a file handle the kdb+ object stored in the file can be obtained by accessing the `value` property of the file handle:

```
>>> _.value
k('0 1 2 3 4 5 6 7 8 9')
```

Now we can delete the data from memory

```
>>> del q.data
```

and load it back from the file using the `pyq.q.load` function:

```
>>> q.load('data')
k('`data')
>>> q.data
k('0 1 2 3 4 5 6 7 8 9')
```

`pyq.q.save` and `pyq.q.load` functions can also take a `pathlib.Path` object

```
>>> data_path = pathlib.Path('data')
>>> q.save(data_path)
k('`:data')
>>> q.load(data_path)
k('`data')
```

It is not necessary to assign data to a global variable before saving it to a file. We can save our 10 integers directly to a file using the `pyq.q.set` function

```
>>> q.set(':0-9', range(10))
k('`:0-9')
```

and read it back using the `pyq.q.set` function

```
>>> q.get(_)
k('0 1 2 3 4 5 6 7 8 9')
```

K objects

The q language has has atoms (scalars), lists, dictionaries, tables and functions. In PyQ, kdb+ objects of any type appear as instances of class `K`. To tell the underlying kdb+ type, one can access the *`type`* property to obtain a type code. For example,

```
>>> vector = q.til(5); scalar = vector.first
>>> vector.type
k('7h')
>>> scalar.type
k('-7h')
```

Basic vector types have type codes in the range 1 through 19 and their elements have the type code equal to the negative of the vector type code. For the basic vector types, one can also get a human readable type name by accessing the *`key`* property:

```
>>> vector.key
k('`long')
```

To get the same from a scalar – convert it to a vector first:

```
>>> scalar.enlist.key
k('`long')
```

Table 1.4: Basic data types

Code	Kdb+ type	Python type
1	`boolean`	`bool`
2	`guid`	`uuid.UUID`
4	`byte`	
5	`short`	
6	`int`	
7	`long`	`int`
8	`real`	
9	`float`	`float`
10	`char`	`bytes` (*)
11	`symbol`	`str`
12	`timestamp`	
13	`month`	
14	`date`	`datetime.date`
16	`timespan`	`datetime.timedelta`
17	`minute`	
18	`second`	
19	`time`	`datetime.time`

(*) Unlike other Python types mentioned in the table above, `bytes` instances get converted to a vector type:

```
>>> K(b'x')
k(',"x"')
>>> q.type(_)
k('10h')
```

There is no scalar character type in Python, so in order to create a `K` character scalar, one will need to use a typed constructor:

```
>>> K.char(b'x')
k('"x"')
```

Typed constructors are discussed in the next section.

Constructors and casts

As we've seen in the previous chapter, it is often not necessary to construct `K` objects explicitly because they are automatically created whenever a Python object is passed to a q function. This is done by passing the Python object to the default `K` constructor.

For example, if you need to pass a type long atom to a q function, you can use a Python `int` instead, but if a different integer type is required, you will need to create it explicitly:

```
>>> K.short(1)
k('1h')
```

Since empty list does not know the element type, passing `[]` to the default `K` constructor produces a generic (type `0h`) list:

```
>>> K([])
k('()')
>>> q.type(_)
k('0h')
```

To create an empty list of a specific type – pass `[]` to one of the named constructors:

```
>>> K.time([])
k('`time$()')
```

Table 1.5: `K` constructors

Constructor	Accepts	Description
`K.boolean()`	`int`, `bool`	logical type `0b` is false and `1b` is true.
`byte()`	`int`, `bytes`	8-bit bytes
`short()`	`int`	16-bit integers
`int()`	`int`	32-bit integers
`long()`	`int`	64-bit integers
`real()`	`int`, `float`	32-bit floating point numbers
`float()`	`int`, `float`	32-bit floating point numbers
`char()`	`str`, `bytes`	8-bit characters
`symbol()`	`str`, `bytes`	interned strings
`timestamp()`	`int` (nanoseconds), `datetime`	date and time
`month()`	`int` (months), `date`	year and month
`date()`	`int` (days), `date`	year, month and day
`datetime()`		deprecated
`timespan()`	`int` (nanoseconds), `timedelta`	duration in nanoseconds
`minute()`	`int` (minutes), `time`	duration or time of day in minutes
`second()`	`int` (seconds), `time`	duration or time of day in seconds
`time()`	`int` (milliseconds), `time`	duration or time of day in milliseconds

The typed constructors can also be used to access infinities an missing values of the given type:

```
>>> K.real.na, K.real.inf
(k('0Ne'), k('0we'))
```

If you already have a `K` object and want to convert it to a different type, you can access the property named after the type name. For example,

```
>>> x = q.til(5)
>>> x.date
k('2000.01.01 2000.01.02 2000.01.03 2000.01.04 2000.01.05')
```

Operators

Both Python and q provide a rich system of operators. In PyQ, `K` objects can appear in many Python expressions where they often behave as native Python objects.

Most operators act on `K` instances as namesake q functions. For example:

```
>>> K(1) + K(2)
k('3')
```

The if statement and boolean operators

Python has three boolean operators `or`, `and` and `not` and `K` objects can appear in boolean expressions. The result of boolean expressions depends on how the objects are tested in Python if statements.

All `K` objects can be tested for "truth". Similarly to the Python numeric types and sequences, `K` atoms of numeric types are true is they are not zero and vectors are true if they are non-empty.

Atoms of non-numeric types follow different rules. Symbols test true except for the empty symbol; characters and bytes tested true except for the null character/byte; guid, timestamp, and (deprecated) datetime types always test as true.

Functions test as true except for the monadic pass-through function:

```
>>> q('::') or q('+') or 1
k('+')
```

Dictionaries and tables are treated as sequences: they are true if non-empty.

Note that in most cases how the object test does not change when Python native types are converted to `K`:

```
>>> objects = [None, 1, 0, True, False, 'x', '', {1:2}, {}, date(2000, 1, 1)]
>>> [bool(o) for o in objects]
[False, True, False, True, False, True, False, True, False, True]
>>> [bool(K(o)) for o in objects]
[False, True, False, True, False, True, False, True, False, True]
```

One exception is the Python `time` type. Starting with version 3.5 all `time` instances test as true, but `time(0)` converts to `k('00:00:00.000')` which tests false:

```
>>> [bool(o) for o in (time(0), K(time(0)))]
[True, False]
```

Note: Python changed the rule for `time(0)` because `time` instances can be timezone aware and because they do not support addition making 0 less than special. Neither of those arguments apply to `q` time, second or minute data types which behave more like `timedelta`.

Arithmetic operations

Python has the four familiar arithmetic operators +, −, ⋆ and / as well as less common ⋆⋆ (exponentiation), % (modulo) and // (floor division). PyQ maps those operators to q "verbs" as follows

Operation	Python	q
addition	+	+
subtraction	−	−
multiplication	⋆	⋆
true division	/	%
exponentiation	⋆⋆	xexp
floor division	//	div
modulo	%	mod

`K` objects can be freely mixed with Python native types in arithmetic expressions and the result is a `K` object in most cases:

```
>>> q.til(10) % 3
k('0 1 2 0 1 2 0 1 2 0')
```

A notable exception occurs when the modulo operator is used for string formatting

```
>>> "%.5f" % K(3.1415)
'3.14150'
```

Unlike python sequences, `K` lists behave very similar to atoms: arithmetic operations act element-wise on them.

Compare

```
>>> [1, 2] * 5
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

and

```
>>> K([1, 2]) * 5
k('5 10')
```

or

```
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]
```

and

```
>>> K([1, 2]) + [3, 4]
k('4 6')
```

The flip (+) operator

The unary + operator acts as *flip()* function on K objects. Applied to atoms, it has no effect:

```
>>> +K(0)
k('0')
```

but it can be used to transpose a matrix:

```
>>> m = K([[1, 2], [3, 4]])
>>> m.show()
1 2
3 4
>>> (+m).show()
1 3
2 4
```

or turn a dictionary into a table:

```
>>> d = q('!', ['a', 'b'], m)
>>> d.show()
a| 1 2
b| 3 4
>>> (+d).show()
a b
---
1 3
2 4
```

Bitwise operators

Python has six bitwise operators: `|`, `^`, `&`, `<<`, `>>`, and `~`. Since there are no bitwise operations in q, PyQ redefines them as follows:

Operation	Result	Notes
`x	y`	element-wise maximum of *x* and *y*	1.
`x ^ y`	*y* with null elements filled with *x*	2.
`x & y`	element-wise minimum of *x* and *y*	1.
`x << n`	*x* shifted left by *n* elements	3.
`x >> n`	*x* shifted right by *n* elements	3.
`~x`	a boolean vector with 1's for zero elements of *x*	

Notes:

1. For boolean vectors, `|` and `&` are also element-wise *or* and *and* operations.
2. For Python integers, the result of `x ^ y` is the bitwise exclusive or. There is no similar operation in `q`, but for boolean vectors exclusive or is equivalent to q <> (not equal).

3. Negative shift counts result in a shift in the opposite direction to that indicated by the operator: `x >> -n` is the same as `x << n`.

Minimum and maximum

Minimum and maximum operators are `&` and `|` in q. PyQ maps similar looking Python bitwise operators to the corresponding q ones:

```
>>> q.til(10) | 5
k('5 5 5 5 5 5 6 7 8 9')
>>> q.til(10) & 5
k('0 1 2 3 4 5 5 5 5 5')
```

The ^ operator

Unlike Python where caret (^) is the binary xor operator, q defines it to denote the fill operation that replaces null values in the right argument with the left argument. PyQ follows the q definition:

```
>>> x = q('1 0N 2')
>>> 0 ^ x
k('1 0 2')
```

The @ operator

Python 3.5 introduced the `@` operator that can be used by user types. Unlike numpy that defines `@` as the matrix multiplication operator, PyQ uses `@` for function application and composition:

```
>>> q.log @ q.exp @ 1
k('1f')
```

Adverbs

Adverbs in q are somewhat similar to Python decorators. They act on functions and produce new functions. The six adverbs are summarized in the table below.

Table 1.6: Adverbs

PyQ	q	Description
`K.each()`	`'`	map or case
`K.over()`	`/`	reduce
`K.scan()`	`\`	accumulate
`K.prior()`	`':`	each-prior
`K.sv()`	`/:`	each-right or scalar from vector
`K.vs()`	`\:`	each-left or vector from scalar

The functionality provided by the first three adverbs is similar to functional programming features scattered throughout Python standard library. Thus `each` is similar to `map()`. For example, given a list of lists of numbers

```
>>> data = [[1, 2], [1, 2, 3]]
```

One can do

```
>>> q.sum.each(data)
k('3 6')
```

or

```
>>> list(map(sum, [[1, 2], [1, 2, 3]]))
[3, 6]
```

and get similar results.

The `over` adverb is similar to the `functools.reduce()` function. Compare

```
>>> q(',').over(data)
k('1 2 1 2 3')
```

and

```
>>> functools.reduce(operator.concat, data)
[1, 2, 1, 2, 3]
```

Finally, the `scan` adverb is similar to the `itertools.accumulate()` function.

```
>>> q(',').scan(data).show()
1 2
1 2 1 2 3
```

```
>>> for x in itertools.accumulate(data, operator.concat):
...     print(x)
...
[1, 2]
[1, 2, 1, 2, 3]
```

Each

The `each` adverb serves double duty in q. When it is applied to a function, it returns a new function that expects lists as arguments and maps the original function over those lists. For example, we can write a "daily return" function in q that takes yesterday's price as the first argument (x), today's price as the second (y) and dividend as the third (z) as follow:

```
>>> r = q('{(y+z-x)%x}')  # Recall that % is the division operator in q.
```

and use it to compute returns from a series of prices and dividends using `r.each`:

```
>>> p = [50.5, 50.75, 49.8, 49.25]
>>> d = [.0, .0, 1.0, .0]
>>> r.each(q.prev(p), p, d)
k('0n 0.004950495 0.0009852217 -0.01104418')
```

When the `each` adverb is applied to an integer vector, it turns the vector v into an n-ary function that for each i-th argument selects its v[i]-th element. For example,

```
>>> v = q.til(3)
>>> v.each([1, 2, 3], 100, [10, 20, 30])
k('1 100 30')
```

Note that scalars passed to `v.each` are treated as infinitely repeated values. Vector arguments must all be of the same length.

Over and scan

Given a function `f`, `f.over` and `f.scan` adverbs are similar as both apply `f` repeatedly, but `f.over` only returns the final result, while `f.scan` returns all intermediate values as well.

For example, recall that the Golden Ratio can be written as a continued fraction as follows

$$\phi = 1 + \frac{1}{1 + \frac{1}{1+\cdots}}$$

or equivalently as the limit of the sequence that can be obtained by starting with 1 and repeatedly applying the function

$$f(x) = 1 + \frac{1}{1+x}$$

The numerical value of the Golden Ratio can be found as

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.618033988749895$$

```
>>> phi = (1+math.sqrt(5)) / 2
>>> phi
1.618033988749895
```

Function f can be written in q as follows:

```
>>> f = q('{1+reciprocal x}')
```

and

```
>>> f.over(1.)
k('1.618034')
```

indeed yields a number recognizable as the Golden Ratio. If instead of `f.over`, we compute `f.scan`, we will get the list of all convergents.

```
>>> x = f.scan(1.)
>>> len(x)
32
```

Note that `f.scan` (and `f.over`) stop calculations when the next iteration yields the same value and indeed `f` applied to the last value returns the same value:

```
>>> f(x.last) == x.last
True
```

which is close to the value computed using the exact formula

```
>>> math.isclose(x.last, phi)
True
```

The number of iterations can be given explicitly by passing two arguments to `f.scan` or `f.over`:

```
>>> f.scan(10, 1.)
k('1 2 1.5 1.666667 1.6 1.625 1.615385 1.619048 1.617647 1.618182 1.617978')
>>> f.over(10, 1.)
k('1.617978')
```

This is useful when you need to iterate a function that does not converge.

Continuing with the Golden Ratio theme, let's define a function

```
>>> f = q('{(last x;sum x)}')
```

that given a pair of numbers returns another pair made out of the last and the sum of the numbers in the original pair. Iterating this function yields the Fibonacci sequence

```
>>> x = f.scan(10,[0, 1])
>>> q.first.each(x)
k('0 1 1 2 3 5 8 13 21 34 55')
```

and the ratios of consecutive Fibonacci numbers form the sequence of Golden Ratio convergents that we've seen before:

```
>>> q.ratios(_)
k('0 0w 1 2 1.5 1.666667 1.6 1.625 1.615385 1.619048 1.617647')
```

Each previous

In the previous section we've seen a function *ratios()* that takes a vector and produces the ratios of the adjacent elements. A similar function called *deltas()* produces the differences between the adjacent elements:

```
>>> q.deltas([1, 3, 2, 5])
k('1 2 -1 3')
```

These functions are in fact implemented in q by applying the `prior` adverb to the division (`%`) and subtraction functions respectively:

```
>>> q.ratios == q('%').prior and q.deltas == q('-').prior
True
```

In general, for any binary function f and a vector v

$$\text{f.prior}(v) = (f(v_1, v_0),\ f(v_2, v_1), \cdots)$$

Adverbs vs and sv

Of all adverbs, these two have the most cryptic names and offer some non-obvious features.

To illustrate how vs and sv modify binary functions, lets give a Python name to the q `,` operator:

```
>>> join = q(',')
```

Suppose you have a list of file names

```
>>> name = K.string(['one', 'two', 'three'])
```

and an extension

```
>>> ext = K.string(".py")
```

You want to append the extension to each name on your list. If you naively call `join` on `name` and `ext`, the result will not be what you might expect:

```
>>> join(name, ext)
k('("one";"two";"three";".";"p";"y")')
```

This happened because `join` treated `ext` as a list of characters rather than an atomic string and created a mixed list of three strings followed by three characters. What we need is to tell `join` to treat its first argument as a vector and the second as a scalar and this is exactly what the `vs` adverb will achieve:

```
>>> join.vs(name, ext)
k('("one.py";"two.py";"three.py")')
```

The mnemonic rule is "vs" = "vector, scalar". Now, if you want to prepend a directory name to each resulting file, you can use the `sv` attribute:

```
>>> d = K.string("/tmp/")
>>> join.sv(d, _)
k('("/tmp/one.py";"/tmp/two.py";"/tmp/three.py")')
```

Input/Output

```
>>> import os
>>> r, w = os.pipe()
>>> h = K(w)(kp("xyz"))
>>> os.read(r, 100)
b'xyz'
>>> os.close(r); os.close(w)
```

Q variables can be accessed as attributes of the 'q' object:

```
>>> q.t = q('([]a:1 2i;b:`x`y)')
>>> sum(q.t.a)
3
>>> del q.t
```

Numeric Computing

NumPy is the fundamental package for scientific computing in Python. NumPy shares common APL ancestry with q and can often operate directly on `K` objects.

Primitive data types

There are eighteen primitive data types in kdb+, eight of those closely match their NumPy analogues and will be called "simple types" in this section. Simple types consist of booleans, bytes, characters, integers of three different sizes, and floating point numbers of two sizes. Seven kdb+ types are dealing with dates, times and durations. Similar data types are available in recent versions of NumPy, but they differ from kdb+ types in many details. Finally, kdb+ symbol, enum and guid types have no direct analogue in NumPy.

Table 1.7: Primitive kdb+ data types as NumPy arrays

No.	kdb+ type	array type	raw	description
1	boolean	bool_	bool_	Boolean (True or False) stored as a byte
2	guid	uint8 (x16)	uint8 (x16)	Globally unique 16-byte identifier
4	byte	uint8	uint8	Byte (0 to 255)
5	short	int16	int16	Signed 16-bit integer
6	int	int32	int32	Signed 32-bit integer
7	long	int64	int64	Signed 64-bit integer
8	real	float32	float32	Single precision 32-bit float
9	float	float64	float64	Double precision 64-bit float
10	char	S1	S1	(byte-)string
11	symbol	str	P	Strings from a pool
12	timestamp	datetime64[ns]	int64	Date and time with nanosecond resolution
13	month	datetime64[M]	int32	Year and month
14	date	datetime64[D]	int32	Date (year, month, day)
16	timespan	timedelta64[ns]	int64	Time duration in nanoseconds
17	minute	datetime64[m]	int32	Time duration (or time of day) in minutes
18	second	datetime64[s]	int32	Time duration (or time of day) in seconds
19	time	datetime64[ms]	int32	Time duration (or time of day) in milliseconds
20+	enum	str	int32	Enumerated strings

Simple types

Kdb+ atoms and vectors of the simple types (booleans, characters, integers and floats) can be viewed as 0- or 1-dimensional NumPy arrays. For example,

```
>>> x = K.real([10, 20, 30])
>>> a = numpy.asarray(x)
>>> a.dtype
dtype('float32')
```

Note that `a` in the example above is not a copy of `x`. It is an array view into the same data:

```
>>> a.base.obj
k('10 20 30e')
```

If you modify `a`, you modify x as well:

```
>>> a[:] = 88
>>> x
k('88 88 88e')
```

Dates, times and durations

An age old question of when to start counting calendar years did not get any easier in the computer age. Python standard `date` starts at

```
>>> date.min
datetime.date(1, 1, 1)
```

more commonly known as

```
>>> date.min.strftime('%B %d, %Y')
'January 01, 0001'
```

and this date is considered to be day 1

```
>>> date.min.toordinal()
1
```

Note that according to the Python calendar the world did not exist before that date:

```
>>> date.fromordinal(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: ordinal must be >= 1
```

At the time of this writing,

```
>>> date.today().toordinal()
736335
```

The designer of kdb+ made a more practical choice for date 0 to be January 1, 2000. As a result, in PyQ we have

```
>>> K.date(0)
k('2000.01.01')
```

and

```
>>> (-2 + q.til(5)).date
k('1999.12.30 1999.12.31 2000.01.01 2000.01.02 2000.01.03')
```

Similarly, the 0 timestamp was chosen to be at midnight of the day 0

```
>>> K.timestamp(0)
k('2000.01.01D00:00:00.000000000')
```

NumPy, however the third choice was made. Kowtowing to the UNIX tradition, NumPy took midnight of January 1, 1970 as the zero mark on its timescales.

```
>>> numpy.array([0], 'datetime64[D]')
array(['1970-01-01'], dtype='datetime64[D]')
>>> numpy.array([0], 'datetime64[ns]')
array(['1970-01-01T00:00:00.000000000'], dtype='datetime64[ns]')
```

PyQ will automatically adjust the epoch when converting between NumPy arrays and `K` objects.

```
>>> d = q.til(2).date
>>> a = numpy.array(d)
>>> d
k('2000.01.01 2000.01.02')
>>> a
array(['2000-01-01', '2000-01-02'], dtype='datetime64[D]')
>>> K(a)
k('2000.01.01 2000.01.02')
```

This convenience comes at a cost of copying the data

```
>>> a[0] = 0
>>> a
array(['1970-01-01', '2000-01-02'], dtype='datetime64[D]')
>>> d
k('2000.01.01 2000.01.02')
```

To avoid such copying, `K` objects can expose their raw data to numpy:

```
>>> b = numpy.asarray(d.data)
>>> b.tolist()
[0, 1]
```

Arrays created this way share their data with the underlying `K` objects. Any change to the array is reflected in kdb+.

```
>>> b[:] += 42
>>> d
k('2000.02.12 2000.02.13')
```

Characters, strings and symbols

Text data appears in kdb+ as character atoms and strings or as symbols and enumerations. Character strings are compatible with NumPy "bytes" type:

```
>>> x = K.string("abc")
>>> a = numpy.asarray(x)
>>> a.dtype.type
<class 'numpy.bytes_'>
```

In the example above, data is shared between the kdb+ string **x** and NumPy array **a**:

```
>>> a[:] = 'x'
>>> x
k('"xxx"')
```

Nested lists

Kdb+ does not have a data type representing multi-dimensional contiguous arrays. In PyQ, a multi-dimensional NumPy array becomes a nested list when passed to `q` functions or converted to `K` objects. For example,

```
>>> a = numpy.arange(12, dtype=float).reshape((2,2,3))
>>> x = K(a)
>>> x
k('((0 1 2f;3 4 5f);(6 7 8f;9 10 11f))')
```

Similarly, kdb+ nested lists of regular shape, become multi-dimensional NumPy arrays when passed to `numpy.array()`:

```
>>> numpy.array(x)
array([[[  0.,   1.,   2.],
        [  3.,   4.,   5.]],

       [[  6.,   7.,   8.],
        [  9.,  10.,  11.]]])
```

Moreover, many NumPy functions can operate directly on kdb+ nested lists, but they internally create a contiguous copy of the data

```
>>> numpy.mean(x, axis=2)
array([[  1.,   4.],
       [  7.,  10.]])
```

Tables and dictionaries

Unlike kdb+ NumPy does not implement column-wise tables. Instead it has record arrays that can store table-like data row by row. PyQ supports two-way conversion between kdb+ tables and NumPy record arrays:

```
>>> trades.show()
sym time  size
--------------
a   09:31 100
a   09:33 300
b   09:32 200
b   09:35 100
```

```
>>> numpy.array(trades)
array([('a', datetime.timedelta(0, 34260), 100),
       ('a', datetime.timedelta(0, 34380), 300),
       ('b', datetime.timedelta(0, 34320), 200),
       ('b', datetime.timedelta(0, 34500), 100)],
      dtype=[('sym', 'O'), ('time', '<m8[m]'), ('size', '<i8')])
```

Enhanced shell

If you have ipython installed in your environment, you can run an interactive IPython shell as follows:

```
$ pyq -m IPython
```

For a better experience, load `pyq.magic` extension:

```
In [1]: %load_ext pyq.magic
```

This makes K objects display nicely in the output and gives you access to the PyQ-specific IPython magic commands:

Line magic `%q`:

```
In [2]: %q ([]a:til 3;b:10*til 3)
Out[2]:
a b
----
0 0
1 10
2 20
```

Cell magic `%%q`:

```
In [4]: %%q
   ....: a: exec a from t where b=20
   ....: b: exec b from t where a=2
   ....: a+b
```

```
   ....:
Out[4]: ,22
```

You can pass following options to the `%%q` cell magic:

-l (dir|script)
: pre-load database or script

-h host:port
: execute on the given host

-o var
: send output to a variable named var

-i var1, .., varN
: input variables

-1
: redirect stdout

-2
: redirect stderr

q) prompt

While in PyQ, you can drop to emulated kdb+ Command Line Interface (CLI). Here is how:

Start pyq:

```
$ pyq
>>> from pyq import q
```

Enter kdb+ CLI:

```
>>> q()
q)t:([]a:til 5; b:10*til 5)
q)t
a b
----
0 0
1 10
2 20
3 30
4 40
```

Exit back to Python:

```
q)\
>>> print("Back to Python")
Back to Python
```

Or you can exit back to shell:

```
q)\\
$
```

Calling Python from KDB+

KDB+ is designed as a platform for multiple programming languages. Out of the box, it comes with q and K distributes variant of ANSI SQL as the "s)" language. Installing pyq gives access to the "p)" language, where "p" obviously stands for "Python". In addition, PyQ provides a mechanism for exporting Python functions to q where they can be called as native q functions.

The "p" language

To access Python from the `q)` prompt, simply start the line with the `p)` prefix and follow with the Python statement(s). Since the standard `q)` prompt does not allow multi-line entries, you are limited to what can be written in one line and need to separate python statements with semicolons.

```
q)p)x = 42; print(x)
42
```

The `p)` prefix can also be used in q scripts. In this case, multi-line python statements can be used as long as additional lines start with one or more spaces. For example, with the following code in hello.q

```
p)def f():
      print('Hello')
p)f()
```

we get

```
$ q hello.q -q
Hello
```

If your script contains more python code than q, you can avoid sprinkling it with `p)`'s by placing the code in a file with .p extension. Thus instead of hello.q described above, we can write the following code in hello.p

```
def f():
    print('Hello')
f()
q.exit(0)
```

and run it the same way:

```
$ q hello.p -q
Hello
```

It is recommended that any substantial amount of Python code be placed in regular python modules or packages with only top level entry points imported and called in q scripts.

Exporting Python functions to q

As we've seen in the previous section, calling python by evaluating "p)" expressions has several limitations. For tighter integration between q and Python, pyq supports exporting Python functions to q. Once exported, python functions appear in q as monadic functions that take a single argument that should be a list. For example, we can make Python's `%`-formatting available in q as follows:

```
>>> def fmt(f, x):
...     return K.string(str(f) % x)
>>> q.fmt = fmt
```

Now, calling the `fmt` function from q will pass the argument list to Python and return the result back to q:

```
q)fmt("%10.6f";acos -1)
"  3.141593"
```

Python functions exported to q should return a `K` object or an instance of one of the simple scalar types: `None`, `bool`, `int`, `float` or `str` which are automatically converted to q `::`, boolean, long, float or symbol respectively.

Exported functions are called from q by supplying a single argument that contains a list of objects to be passed to the Python functions as `K`-valued arguments.

Note: To pass a single argument to an exported function, it has to be enlisted. For example,

```
q)p)q.erf = math.erf
q)erf enlist 1
0.8427008
```

Reference Manual

(This section is generated from the PyQ source code. You can access most of this material using pydoc or the built-in help method.)

K
q

class K

K.**abs**()
: absolute value function

  For details, see *q.abs* and abs on code.kx.com.

K.**acos**()
: arc cosine function

  For details, see *q.acos* and acos on code.kx.com.

K.**aj**()
: asof join function

  For details, see *q.aj* and aj on code.kx.com.

K.**aj0**()
: asof join function

  For details, see *q.aj0* and aj on code.kx.com.

K.**all**()
: all function

  For details, see *q.all* and all on code.kx.com.

K.**and_**()
: and verb

  For details, see *q.and_* and and on code.kx.com.

K.**any**()
: any function

  For details, see *q.any* and any on code.kx.com.

K.**asc**()
: ascending function

  For details, see *q.asc* and asc on code.kx.com.

K.**asin**()
: arc sine function

  For details, see *q.asin* and asin on code.kx.com.

K.**asof**()
: asof verb

  For details, see *q.asof* and asof on code.kx.com.

K.**atan**()
: arc tangent function

  For details, see *q.atan* and atan on code.kx.com.

K.**attr**()
: attributes function

  For details, see *q.attr* and attr on code.kx.com.

K.**avg**()
: average function

  For details, see *q.avg* and avg on code.kx.com.

K.**avgs**()
: running averages function

  For details, see *q.avgs* and avgs on code.kx.com.

K.**bin**()
: binary search verb

  For details, see *q.bin* and bin on code.kx.com.

K.**binr**()
: binary search verb

  For details, see *q.binr* and bin on code.kx.com.

K.**ceiling**()
: ceiling function

  For details, see *q.ceiling* and ceiling on code.kx.com.

K.**cols**()
: columns function

  For details, see *q.cols* and cols on code.kx.com.

K.**cor**()
: correlation verb

  For details, see *q.cor* and cor on code.kx.com.

K.**cos**()
cosine function

For details, see *q.cos* and cos on code.kx.com.

K.**count**()
count function

For details, see *q.count* and count on code.kx.com.

K.**cov**()
covariance verb

For details, see *q.cov* and cov on code.kx.com.

K.**cross**()
cross product verb

For details, see *q.cross* and cross on code.kx.com.

K.**csv**()
csv global

For details, see *q.csv* and csv on code.kx.com.

K.**cut**()
cut verb

For details, see *q.cut* and cut on code.kx.com.

K.**deltas**()
deltas function

For details, see *q.deltas* and deltas on code.kx.com.

K.**desc**()
descending sort function

For details, see *q.desc* and desc on code.kx.com.

K.**dev**()
standard deviation function

For details, see *q.dev* and dev on code.kx.com.

K.**differ**()
differ function

For details, see *q.differ* and differ on code.kx.com.

K.**distinct**()
distinct function

For details, see *q.distinct* and distinct on code.kx.com.

K.**div**()
integer division verb

For details, see *q.div* and div on code.kx.com.

K.**dsave**()
dsave function

For details, see *q.dsave* and dsave on code.kx.com.

K.**each**()
each adverb

For details, see *q.each* and each on code.kx.com.

K.**ej**()
equijoin verb

For details, see *q.ej* and ej on code.kx.com.

K.**ema**()
exponentially weighted moving average verb

For details, see *q.ema* and ema on code.kx.com.

K.**ema**()
exponentially weighted moving average verb

For details, see *q.ema* and ema on code.kx.com.

K.**enlist**()
enlist function

For details, see *q.enlist* and enlist on code.kx.com.

K.**eval**()
eval function

For details, see *q.eval* and eval on code.kx.com.

K.**except_**()
except verb

For details, see *q.except_* and except on code.kx.com.

K.**exp**()
exp function

For details, see *q.exp* and exp on code.kx.com.

K.**fby**()
filter-by

For details, see *q.fby* and fby on code.kx.com.

K.**fills**()
fills function

For details, see *q.fills* and fills on code.kx.com.

K.**first**()
first function

For details, see *q.first* and first on code.kx.com.

K.**fkeys**()
fkeys function

For details, see *q.fkeys* and fkeys on code.kx.com.

K.**flip**()
flip function

For details, see *q.flip* and flip on code.kx.com.

K.**floor**()
floor function

For details, see *q.floor* and floor on code.kx.com.

K.**get**()
get function

For details, see *q.get* and get on code.kx.com.

K.**getenv**()
getenv function

For details, see *q.getenv* and getenv on code.kx.com.

K.**group**()
group function

For details, see *q.group* and group on code.kx.com.

K.**gtime**()
gtime function

For details, see *q.gtime* and gtime on code.kx.com.

K.**hclose**()
hclose function

For details, see *q.hclose* and hclose on code.kx.com.

K.**hcount**()
hcount function

For details, see *q.hcount* and hcount on code.kx.com.

K.**hdel**()
hdel function

For details, see *q.hdel* and hdel on code.kx.com.

K.**hopen**()
hopen function

For details, see *q.hopen* and hopen on code.kx.com.

K.**hsym**()
hsym function

For details, see *q.hsym* and hsym on code.kx.com.

K.**iasc**()
ascending function

For details, see *q.iasc* and iasc on code.kx.com.

K.**idesc**()
descending function

For details, see *q.idesc* and idesc on code.kx.com.

K.**ij**()
inner join verb

For details, see *q.ij* and ij on code.kx.com.

K.**ijf**()
The ijf function.

For details, see *q.ijf* and ijf on code.kx.com.

K.**in_**()
membership verb

For details, see *q.in_* and in on code.kx.com.

K.**insert**()
insert verb

For details, see *q.insert* and insert on code.kx.com.

K.**inter**()
intersect verb

For details, see *q.inter* and inter on code.kx.com.

K.**inv**()
inverse function

For details, see *q.inv* and inv on code.kx.com.

K.**key**()
key function

For details, see *q.key* and key on code.kx.com.

K.**keys**()
keys function

For details, see *q.keys* and keys on code.kx.com.

K.**last**()
last function

For details, see *q.last* and last on code.kx.com.

K.**like**()
pattern matching verb

For details, see *q.like* and like on code.kx.com.

K.**lj**()
left join verb

For details, see *q.lj* and lj on code.kx.com.

K.**ljf**()
The ljf function.

For details, see *q.ljf* and ljf on code.kx.com.

K.**load**()
load function

For details, see *q.load* and load on code.kx.com.

K.**log**()
log function

For details, see *q.log* and log on code.kx.com.

K.**lower**()
lowercase function

For details, see *q.lower* and lower on code.kx.com.

K.**lsq**()
least squares verb

For details, see *q.lsq* and lsq on code.kx.com.

K.**ltime**()
ltime function

For details, see *q.ltime* and ltime on code.kx.com.

K.**ltrim**()
left trim function

For details, see *q.ltrim* and ltrim on code.kx.com.

K.**mavg**()
moving average verb

For details, see *q.mavg* and mavg on code.kx.com.

K.**max**()
maximum function

For details, see *q.max* and max on code.kx.com.

K.**maxs**()
maximums function

For details, see *q.maxs* and maxs on code.kx.com.

K.**mcount**()
moving count verb

For details, see *q.mcount* and mcount on code.kx.com.

K.**md5**()
md5 function

For details, see *q.md5* and md5 on code.kx.com.

K.**mdev**()
moving deviation verb

For details, see *q.mdev* and mdev on code.kx.com.

K.**med**()
median function

For details, see *q.med* and med on code.kx.com.

K.**meta**()
meta data function

For details, see *q.meta* and meta on code.kx.com.

K.**min**()
minimum function

For details, see *q.min* and min on code.kx.com.

K.**mins**()
minimums function

For details, see *q.mins* and mins on code.kx.com.

K.**mmax**()
moving maximum verb

For details, see *q.mmax* and mmax on code.kx.com.

K.**mmin**()
moving minimum verb

For details, see *q.mmin* and mmin on code.kx.com.

K.**mmu**()
matrix multiplication verb

For details, see *q.mmu* and mmu on code.kx.com.

K.**mod**()
modulus verb

For details, see *q.mod* and mod on code.kx.com.

K.**msum**()
moving sum verb

For details, see *q.msum* and msum on code.kx.com.

K.**neg**()
negative function

For details, see *q.neg* and neg on code.kx.com.

K.**next**()
next function

For details, see *q.next* and next on code.kx.com.

K.**not_**()
logical not function

For details, see *q.not_* and not on code.kx.com.

K.**null**()
null function

For details, see *q.null* and null on code.kx.com.

K.**or_**()
or verb

For details, see *q.or_* and or on code.kx.com.

K.**over**()
over adverb

For details, see *q.over* and over on code.kx.com.

K.**parse**()
parse function

For details, see *q.parse* and parse on code.kx.com.

K.**peach**()
parallel each adverb

For details, see *q.peach* and peach on code.kx.com.

K.**pj**()
plus join verb

For details, see *q.pj* and pj on code.kx.com.

K.**prd**()
product function

For details, see *q.prd* and prd on code.kx.com.

K.**prds**()
cumulative product function

For details, see *q.prds* and prds on code.kx.com.

K.**prev**()
prev function

For details, see *q.prev* and prev on code.kx.com.

K.**prior**()
prior function

For details, see *q.prior* and prior on code.kx.com.

K.**rand**()
random function

For details, see *q.rand* and rand on code.kx.com.

K.**rank**()
rank function

For details, see *q.rank* and rank on code.kx.com.

K.**ratios**()
ratios function

For details, see *q.ratios* and ratios on code.kx.com.

K.**raze**()
raze function

For details, see *q.raze* and raze on code.kx.com.

K.**read0**()
file read function

For details, see *q.read0* and read0 on code.kx.com.

K.**read1**()
file read function

For details, see *q.read1* and read1 on code.kx.com.

K.**reciprocal**()
reciprocal function

For details, see *q.reciprocal* and reciprocal on code.kx.com.

K.**reval**()
reval function

For details, see *q.reval* and reval on code.kx.com.

K.**reval**()
reval function

For details, see *q.reval* and reval on code.kx.com.

K.**reverse**()
reverse function

For details, see *q.reverse* and reverse on code.kx.com.

K.**rload**()
rload function

For details, see *q.rload* and rload on code.kx.com.

K.**rotate**()
rotate verb

For details, see *q.rotate* and rotate on code.kx.com.

K.**rsave**()
rsave function

For details, see *q.rsave* and rsave on code.kx.com.

K.**rtrim**()
right trim function

For details, see *q.rtrim* and rtrim on code.kx.com.

K.**save**()
save function

For details, see *q.save* and save on code.kx.com.

K.**scan**()
scan adverb

For details, see *q.scan* and scan on code.kx.com.

K.**scov**()
statistical covariance verb

For details, see *q.scov* and scov on code.kx.com.

K.**scov**()
statistical covariance verb

For details, see *q.scov* and scov on code.kx.com.

K.**sdev**()
statistical standard deviation function

For details, see *q.sdev* and sdev on code.kx.com.

K.**sdev**()
statistical standard deviation function

For details, see *q.sdev* and sdev on code.kx.com.

K.**set**()
set verb

For details, see *q.set* and set on code.kx.com.

K.**setenv**()
setenv verb

For details, see *q.setenv* and setenv on code.kx.com.

K.**show**()
show function

For details, see *q.show* and show on code.kx.com.

K.**signum**()
signum function

For details, see *q.signum* and signum on code.kx.com.

K.**sin**()
sine function

For details, see *q.sin* and sin on code.kx.com.

K.**sqrt**()
square root function

For details, see *q.sqrt* and sqrt on code.kx.com.

K.**ss**()
string search function

For details, see *q.ss* and ss on code.kx.com.

K.**ssr**()
string search replace function

For details, see *q.ssr* and ssr on code.kx.com.

K.**string**()
string function

For details, see *q.string* and string on code.kx.com.

K.**sublist**()
sublist verb

For details, see *q.sublist* and sublist on code.kx.com.

K.**sum**()
sum function

For details, see *q.sum* and sum on code.kx.com.

K.**sums**()
cumulative sum function

For details, see *q.sums* and sums on code.kx.com.

K.**sv**()
scalar from vector verb

For details, see *q.sv* and sv on code.kx.com.

K.**svar**()
statistical variance function

For details, see *q.svar* and svar on code.kx.com.

K.**svar**()
statistical variance function

For details, see *q.svar* and svar on code.kx.com.

K.**system**()
system command function

For details, see *q.system* and system on code.kx.com.

K.**tables**()
tables function

For details, see *q.tables* and tables on code.kx.com.

K.**tan**()
tangent function

For details, see *q.tan* and tan on code.kx.com.

K.**til**()
til function

For details, see *q.til* and til on code.kx.com.

K.**trim**()
trim function

For details, see *q.trim* and trim on code.kx.com.

K.**type**()
type function

For details, see *q.type* and type on code.kx.com.

K.**uj**()
union join verb

For details, see *q.uj* and uj on code.kx.com.

K.**ungroup**()
ungroup function

For details, see *q.ungroup* and ungroup on code.kx.com.

K.**union**()
union verb

For details, see *q.union* and union on code.kx.com.

K.**upper**()
uppercase function

For details, see *q.upper* and upper on code.kx.com.

K.**upsert**()
upsert verb

For details, see *q.upsert* and upsert on code.kx.com.

K.**value**()
value function

For details, see *q.value* and value on code.kx.com.

K.**var**()
variance function

For details, see *q.var* and var on code.kx.com.

K.**view**()
view function

For details, see *q.view* and view on code.kx.com.

K.**views**()
views function

For details, see *q.views* and views on code.kx.com.

K.**vs**()
vector from scalar verb

For details, see *q.vs* and vs on code.kx.com.

K.**wavg**()
weighted average verb

For details, see *q.wavg* and wavg on code.kx.com.

K.**where**()
where function

For details, see *q.where* and where on code.kx.com.

K.**within**()
within verb

For details, see *q.within* and within on code.kx.com.

K.**wj**()
window join function

For details, see *q.wj* and wj on code.kx.com.

K.**wj1**()
The wj1 function.

For details, see *q.wj1* and wj1 on code.kx.com.

K.**wsum**()
weighted sum verb

For details, see *q.wsum* and wsum on code.kx.com.

K.**ww**()
The ww function.

For details, see *q.ww* and ww on code.kx.com.

K.**xasc**()
ascending sort verb

For details, see *q.xasc* and xasc on code.kx.com.

K.**xbar**()
interval bar verb

For details, see *q.xbar* and xbar on code.kx.com.

K.**xcol**()
rename columns verb

For details, see *q.xcol* and xcol on code.kx.com.

K.**xcols**()
reorder columns verb

For details, see *q.xcols* and xcols on code.kx.com.

K.**xdesc**()
descending sort verb

For details, see *q.xdesc* and xdesc on code.kx.com.

K.**xexp**()
power verb

For details, see *q.xexp* and xexp on code.kx.com.

K.**xgroup**()
grouping verb

For details, see *q.xgroup* and xgroup on code.kx.com.

K.**xkey**()
set primary key verb

For details, see *q.xkey* and xkey on code.kx.com.

K.**xlog**()
base-x log verb

For details, see *q.xlog* and xlog on code.kx.com.

K.**xprev**()
previous verb

For details, see *q.xprev* and xprev on code.kx.com.

K.**xrank**()
buckets verb

For details, see *q.xrank* and xrank on code.kx.com.

namespace q

pyq.**q**

Q functions

q.**abs**()
absolute value function The abs function computes the absolute value of its argument. Null is returned if the argument is null.

```
>>> q.abs([-1, 0, 1, None])
k('1 0 1 0N')
```

See also abs on code.kx.com.

q.**acos**()
arc cosine function

See also acos on code.kx.com.

q.**aj**()
asof join function

See also aj on code.kx.com.

q.**aj0**()
asof join function

See also aj on code.kx.com.

q.**all**()
all function

See also all on code.kx.com.

q.**and_**()
and verb

See also and on code.kx.com.

q.**any**()
any function

See also any on code.kx.com.

q.**asc**()
ascending function

See also asc on code.kx.com.

q.**asin**()
arc sine function

See also asin on code.kx.com.

q.**asof**()
asof verb

See also asof on code.kx.com.

q.**atan**()
arc tangent function

See also atan on code.kx.com.

q.**attr**()
attributes function

See also attr on code.kx.com.

q.**avg**()
average function

See also avg on code.kx.com.

q.**avgs**()
running averages function

See also avgs on code.kx.com.

q.**bin**()
binary search verb

See also bin on code.kx.com.

q.**binr**()
binary search verb

See also bin on code.kx.com.

q.**ceiling**()
ceiling function

See also ceiling on code.kx.com.

q.**cols**()
columns function

See also cols on code.kx.com.

q.**cor**()
correlation verb

See also cor on code.kx.com.

q.**cos**()
cosine function

See also cos on code.kx.com.

q.**count**()
count function

See also count on code.kx.com.

q.**cov**()
covariance verb

See also cov on code.kx.com.

q.**cross**()
cross product verb

See also cross on code.kx.com.

q.**csv**()
csv global

See also csv on code.kx.com.

q.**cut**()
cut verb

See also cut on code.kx.com.

q.**deltas**()
deltas function

See also deltas on code.kx.com.

q.**desc**()
descending sort function

See also desc on code.kx.com.

q.**dev**()
standard deviation function

See also dev on code.kx.com.

q.**differ**()
differ function

See also differ on code.kx.com.

q.**distinct**()
distinct function

See also distinct on code.kx.com.

q.**div**()
integer division verb

See also div on code.kx.com.

q.**dsave**()
dsave function

See also dsave on code.kx.com.

q.**each**()
each adverb

See also each on code.kx.com.

q.**ej**()
equijoin verb

See also ej on code.kx.com.

q.**ema**()
exponentially weighted moving average verb

See also ema on code.kx.com.

q.**ema**()
exponentially weighted moving average verb

See also ema on code.kx.com.

q.**enlist**()
enlist function

See also enlist on code.kx.com.

q.**eval**()
eval function

See also eval on code.kx.com.

q.**except_**()
except verb

See also except on code.kx.com.

q.**exp**()
exp function

See also exp on code.kx.com.

q.**fby**()
filter-by

See also fby on code.kx.com.

q.**fills**()
fills function

See also fills on code.kx.com.

q.**first**()
first function

See also first on code.kx.com.

q.**fkeys**()
fkeys function

See also fkeys on code.kx.com.

q.**flip**()
flip function

See also flip on code.kx.com.

q.**floor**()
floor function

See also floor on code.kx.com.

q.**get**()
get function

See also get on code.kx.com.

q.**getenv**()
getenv function

See also getenv on code.kx.com.

q.**group**()
group function

See also group on code.kx.com.

q.**gtime**()
gtime function

See also gtime on code.kx.com.

q.**hclose**()
hclose function

See also hclose on code.kx.com.

q.**hcount**()
hcount function

See also hcount on code.kx.com.

q.**hdel**()
hdel function

See also hdel on code.kx.com.

q.**hopen**()
hopen function

See also hopen on code.kx.com.

q.**hsym**()
hsym function

See also hsym on code.kx.com.

q.**iasc**()
ascending function

See also iasc on code.kx.com.

q.**idesc**()
descending function

See also idesc on code.kx.com.

q.**ij**()
inner join verb

See also ij on code.kx.com.

q.**ijf**()
The ijf function.

See also ijf on code.kx.com.

q.**in_**()
membership verb

See also in on code.kx.com.

q.**insert**()
insert verb

See also insert on code.kx.com.

q.**inter**()
intersect verb

See also inter on code.kx.com.

q.**inv**()
inverse function

See also inv on code.kx.com.

q.**key**()
key function

See also key on code.kx.com.

q.**keys**()
keys function

See also keys on code.kx.com.

q.**last**()
last function

See also last on code.kx.com.

q.**like**()
pattern matching verb

See also like on code.kx.com.

q.**lj**()
left join verb

See also lj on code.kx.com.

q.**ljf**()
The ljf function.

See also ljf on code.kx.com.

q.**load**()
load function

See also load on code.kx.com.

q.**log**()
log function

See also log on code.kx.com.

q.**lower**()
lowercase function

See also lower on code.kx.com.

q.**lsq**()
least squares verb

See also lsq on code.kx.com.

q.**ltime**()
ltime function

See also ltime on code.kx.com.

q.**ltrim**()
left trim function

See also ltrim on code.kx.com.

q.**mavg**()
moving average verb

See also mavg on code.kx.com.

q.**max**()
maximum function

See also max on code.kx.com.

q.**maxs**()
maximums function

See also maxs on code.kx.com.

q.**mcount**()
moving count verb

See also mcount on code.kx.com.

q.**md5**()
: md5 function

  See also md5 on code.kx.com.

q.**mdev**()
: moving deviation verb

  See also mdev on code.kx.com.

q.**med**()
: median function

  See also med on code.kx.com.

q.**meta**()
: meta data function

  See also meta on code.kx.com.

q.**min**()
: minimum function

  See also min on code.kx.com.

q.**mins**()
: minimums function

  See also mins on code.kx.com.

q.**mmax**()
: moving maximum verb

  See also mmax on code.kx.com.

q.**mmin**()
: moving minimum verb

  See also mmin on code.kx.com.

q.**mmu**()
: matrix multiplication verb

  See also mmu on code.kx.com.

q.**mod**()
: modulus verb

  See also mod on code.kx.com.

q.**msum**()
: moving sum verb

  See also msum on code.kx.com.

q.**neg**()
: negative function

  See also neg on code.kx.com.

q.**next**()
: next function

  See also next on code.kx.com.

q.**not_**()
logical not function

See also not on code.kx.com.

q.**null**()
null function

See also null on code.kx.com.

q.**or_**()
or verb

See also or on code.kx.com.

q.**over**()
over adverb

See also over on code.kx.com.

q.**parse**()
parse function

See also parse on code.kx.com.

q.**peach**()
parallel each adverb

See also peach on code.kx.com.

q.**pj**()
plus join verb

See also pj on code.kx.com.

q.**prd**()
product function

See also prd on code.kx.com.

q.**prds**()
cumulative product function

See also prds on code.kx.com.

q.**prev**()
prev function

See also prev on code.kx.com.

q.**prior**()
prior function

See also prior on code.kx.com.

q.**rand**()
random function

See also rand on code.kx.com.

q.**rank**()
rank function

See also rank on code.kx.com.

q.**ratios**()
ratios function

See also ratios on code.kx.com.

q.**raze**()
raze function

See also raze on code.kx.com.

q.**read0**()
file read function

See also read0 on code.kx.com.

q.**read1**()
file read function

See also read1 on code.kx.com.

q.**reciprocal**()
reciprocal function

See also reciprocal on code.kx.com.

q.**reval**()
reval function

See also reval on code.kx.com.

q.**reval**()
reval function

See also reval on code.kx.com.

q.**reverse**()
reverse function

See also reverse on code.kx.com.

q.**rload**()
rload function

See also rload on code.kx.com.

q.**rotate**()
rotate verb

See also rotate on code.kx.com.

q.**rsave**()
rsave function

See also rsave on code.kx.com.

q.**rtrim**()
right trim function

See also rtrim on code.kx.com.

q.**save**()
save function

See also save on code.kx.com.

q.**scan**()
scan adverb

See also scan on code.kx.com.

q.**scov**()
statistical covariance verb

See also scov on code.kx.com.

q.**scov**()
statistical covariance verb

See also scov on code.kx.com.

q.**sdev**()
statistical standard deviation function

See also sdev on code.kx.com.

q.**sdev**()
statistical standard deviation function

See also sdev on code.kx.com.

q.**set**()
set verb

See also set on code.kx.com.

q.**setenv**()
setenv verb

See also setenv on code.kx.com.

q.**show**()
show function

See also show on code.kx.com.

q.**signum**()
signum function

See also signum on code.kx.com.

q.**sin**()
sine function

See also sin on code.kx.com.

q.**sqrt**()
square root function

See also sqrt on code.kx.com.

q.**ss**()
string search function

See also ss on code.kx.com.

q.**ssr**()
string search replace function

See also ssr on code.kx.com.

q.**string**()
string function

See also string on code.kx.com.

q.**sublist**()
sublist verb

See also sublist on code.kx.com.

q.**sum**()
sum function

See also sum on code.kx.com.

q.**sums**()
cumulative sum function

See also sums on code.kx.com.

q.**sv**()
scalar from vector verb

See also sv on code.kx.com.

q.**svar**()
statistical variance function

See also svar on code.kx.com.

q.**svar**()
statistical variance function

See also svar on code.kx.com.

q.**system**()
system command function

See also system on code.kx.com.

q.**tables**()
tables function

See also tables on code.kx.com.

q.**tan**()
tangent function

See also tan on code.kx.com.

q.**til**()
til function

See also til on code.kx.com.

q.**trim**()
trim function

See also trim on code.kx.com.

q.**type**()
type function

See also type on code.kx.com.

q.**uj**()
union join verb

See also uj on code.kx.com.

q.**ungroup**()
ungroup function

See also ungroup on code.kx.com.

q.**union**()
union verb

See also union on code.kx.com.

q.**upper**()
uppercase function

See also upper on code.kx.com.

q.**upsert**()
upsert verb

See also upsert on code.kx.com.

q.**value**()
value function

See also value on code.kx.com.

q.**var**()
variance function

See also var on code.kx.com.

q.**view**()
view function

See also view on code.kx.com.

q.**views**()
views function

See also views on code.kx.com.

q.**vs**()
vector from scalar verb

See also vs on code.kx.com.

q.**wavg**()
weighted average verb

See also wavg on code.kx.com.

q.**where**()
where function

See also where on code.kx.com.

q.**within**()
within verb

See also within on code.kx.com.

q.**wj**()
window join function

See also wj on code.kx.com.

q.**wj1**()
The wj1 function.

See also wj1 on code.kx.com.

q.**wsum**()
weighted sum verb

See also wsum on code.kx.com.

q.**ww**()
The ww function.

See also ww on code.kx.com.

q.**xasc**()
ascending sort verb

See also xasc on code.kx.com.

q.**xbar**()
interval bar verb

See also xbar on code.kx.com.

q.**xcol**()
rename columns verb

See also xcol on code.kx.com.

q.**xcols**()
reorder columns verb

See also xcols on code.kx.com.

q.**xdesc**()
descending sort verb

See also xdesc on code.kx.com.

q.**xexp**()
power verb

See also xexp on code.kx.com.

q.**xgroup**()
grouping verb

See also xgroup on code.kx.com.

q.**xkey**()
set primary key verb

See also xkey on code.kx.com.

q.**xlog**()
base-x log verb

See also xlog on code.kx.com.

q.**xprev**()
previous verb

See also xprev on code.kx.com.

q.**xrank**()
buckets verb

See also xrank on code.kx.com.

Version History

PyQ 4.0.1

Released on 2017-03-15

Enhancements:

- !509 - #903: Fixed a reference leak in debug build and a gcc 4.8.5 compiler warning.
- !505 - #901: Provide a fallback for systems that lack CPU_COUNT, e.g. RHEL 5.
- !502 - #899: Corrected integer types on 32-bit systems and added explicit casts when necessary.

Documentation:

- !511 - Use locally stored intersphinx inventory.
- !506 - #902 Updated README.

PyQ 4.0

Released on 2017-03-02

New Features:

- !365 - #756: Expose okx from k.h in Python.
- !376 - #806: Hooked basic prompt toolkit functionality into cmdtloop.
- !384 - #809: Implemented the qp script - like pq but start at the q) prompt.
- !385 - #806: Add bottom toolbar to q) prompt.
- !378 - #809: Implemented ipyq and pq scripts.
- !387 - #813: Implemented the @ operator.
- !401 - #828: Implemented type-0 list to array conversions.
- !402 - #775: Implemented getitem for enumerated lists.
- !404 - #833: Implemented *K.___sizeof__()* method.
- !359 - #642: Implement typed constructors and casts
- !390 - #815: Implemented the data attribute for the K objects in C.
- !396 - #829: Implemented basic nd > 1 case: C contiguous and simple type.
- !410 - #840: Implemented shift operators.
- !420 - #851: Implemented setm() and m9() in _k.

- !422 - #852: Implemented conversion from arbitrary sequences to K.
- !428 - #835: Implemented *K.__rmatmul__*.
- !432 - #856: Implemented file system path protocol for file handles.
- !435 - #598: Added support for pathlib2.
- !437 - #855: Added support for complex numbers.
- !439 - #791: Implemented _n attribute for K objects.
- !467 - #873: Implement K.timespan(int) constructor

Enhancements:

- !297 - #752: More datetime64 to q conversions
- !314 - #672: Improve calling Python functions from q
- !315 - #766: Defined the *__dir__* method for class _Q.
- !316 - #767: Make "exec" method callable without trailing _ in PY3K
- !330 - #779: Reimplemented new and call in C
- !352 - #792: Restore support for KXVER=2.
- !354 - #796: Conversion of "small" kdb+ longs will now produce Python ints under Python 2.x.
- !355 - #769: Restore array struct
- !358 - #798: Revisit array to k conversions.
- !375 - #791: K object attributes
- !377 - #807: Clean up and reuse the list of q functions between K and q
- !379 - #808: Clean up pyq namespace
- !380 - #791: Replaced .inspect(b't') with ._t.
- !381 - #806: Return to Python prompt when Control-D or Control-C is pressed.
- !382 - #659: Get rid of KXVER in the C module name.
- !383 - #810: Clean up q namespace
- !388 - #779, #798: Removed unused variables.
- !389 - #818: Use fully qualified name for the internal K base class.
- !391 - #816: temporal data lists to array conversion
- !394 - #823: Preload kdb+ database if provided on pyq command line.
- !397 - #830: Make sure strings obtained from q symbols are interned.
- !398 - #806: Added a simple word completer.
- !399 - #819: Make K.string accept unicode in Python 2.x and bytes in Python 3.x.
- !400 - #806: Clean python exit on \
- !405 - #836: Reimplemented *K.__bool__* in C.
- !406 - #837: Reimplemented *K.__get__* in C.
- !408 - #838: Install sphinxcontrib-spelling package in the deploy stage.
- !413 - #842: K to bytes conversion

- !423 - #852: Added special treatment of symbols in _from_sequence(); allow mixed lists in conversions.
- !424 - #852: Fixed the case of empty sequence. Use K._from_sequence as a tuple converter.
- !425 - #852: Remove dict workaround
- !426 - #853: Make dict[i] consistent with list[i]
- !429 - #854: Walk up the mro to discover converters
- !430 - #608: Return K from mixed K - numpy array operations.
- !431 - #679: Fixed conversion of enumeration scalars into strings.
- !442 - #808: pyq globals clean-up
- !443 - #858: The "nil" object does not crash show() anymore.
- !444 - #817: Clip int(q('0N')) to -0W when building K.long lists.
- !445 - #857: Adverbs revisited
- !446 - #861: Allow unary and binary ops and projections to be called with keywords.
- !447 - #857: Use vs (sv) instead of each_left(right).
- !449 - #864: Corrected the date bounds and added a comprehensive test.
- !450 - #865: Fixed x.char cast
- !455 - #863: Allow out-of-range scalar dates to be converted to ±0Wd.
- !460 - #870: K.timestamp bug
- !470 - #874: K.boolean redesign
- !477 - #875: Make sure bool(enum scalar) works in various exotic scenarios.
- !481 - #881: K._ja bug
- !483 - #850: Use py2x converters in atom constructors.
- !485 - #882: Return 0w on overflow
- !486 - #883: Make boolean constructor stricter : Allow only integer-like values in K._kb().
- !487 - #884: Detect mappings in typed constructors.
- !490 - #841: Fixed mv_release.
- !492 - #886: Fix two bugs in pyq executable; improve setup tests
- !494 - #891: Fix crash in K._kc()

CI and tests improvements:

- !349, !456, !456, !471, !457, !459, !464 - #695, #793, #867: Improvements in code coverage reporting.
- !350 - #794: Run pycodestyle in tox.
- !411 - #827: Use Python 3.6 and 2.7.13 in CI.
- !415, !451 - #845: Use Docker for CI
- !433 - #679: Fixed test on kdb+ 2.x.
- !436 - Add numpy 1.12 to the CI tests.
- !440 - #803: keywords and descriptions from code.kx.com.
- !452 - Add kdb+ 3.5t to the CI tests.

- !461 - #866: Added tests and fixed timestamp range.
- !475 - Use random CPU and limit one CPU core per job in CI.
- !489 - #885: Reformatted code in test files.
- !318, !351, !474, !478, !479, !480, !484, !488, !491 - #768: Improve C code test coverage.

Documentation:

- !341 - #789: Updated README: Test section.
- !353 - #764: simpler docstrings
- !360 - #764: Reorganized documentation. Minor fixes.
- !361 - #764: More docs improvements
- !362 - #764: docs improvements
- !366 - #764: test docs build in tox
- !371 - #803: Updated 32-bit Python/PyQ guide to use Python 3.6.
- !374 - #804: doc style improvements
- !373 - #764 and #777 table to array and sphinx doctest
- !392 - #820: What's New in 4.0
- !403 - #832: spellcheck docs
- !407 - #838: Add doc path to sys.path in conf.py.
- !409 - #803 Docs additions
- !412 - #803: Make documentation testing a separate stage.
- !427 - #803: more docs
- !448 - #803: More docs
- !469 - #871: More docs
- !438 - #854 (#820): Added a what's new entry about named tuples conversion.
- !472 - #803: Added adverbs documentation
- !493 - #803: Document calling Python from q
- !462, !463, !465, !468, !473 - Logo improvements

Setup:

- !337 - #782: Use install extras to install requirements.
- !339 - #782: Use extras instead of deps in tox.ini.
- !340 - #788: Add ipython extras.

PyQ 3.8.4

Released on 2017-01-13

- !414 - #843: Setup should not fail if VIRTUAL_ENV is undefined
- !395 - #825: Fixed uninitialized "readonly" field in getbuffer

PyQ 3.8.3

Released on 2016-12-15

- !357 - #799: Several documentation fixes.
- !368 - #802: Setup should not fail if $VIRTUAL_ENV/q does not exist.

PyQ 3.8.2

Released on 2016-12-01

Documentation improvements:

- !306 - #763: Update README.md - fixed INSTALL link.
- !312 - Fix formatting; ?? -> date of the release in the CHANGELOG.
- !322 - Fixed formatting error in the documentation.
- !324 - #744: use pip to install from the source.
- !338 - #785: Virtual environment setup guide.
- !346 - #764: docs improvements
- !342 - #787: Added links to rtd documentation.

PyQ executable improvements:

- !310 - #761: Allow PyQ executable to be compiled as 32-bit on 64-bit platform.
- !329 - #646: Print PyQ, KDB+ and Python versions if –versions option is given to pyq.
- !332 - #646: Print full PyQ version.
- !333 - #781: Find QHOME when q is installed next to bin/pyq but no venv is set.
- !336 - #783: Fixed a bug in CPUS processing
- !345 - #646: Added NumPy version to –versions output.

Other improvements and bug fixes:

- !308 - #759: Return an empty slice when (stop - start) // stride < 0.
- !320 - #771: Workaround for OrderedDict bug in Python 3.5
- !323 - #773: Renamed ipython into jupyter; added starting notebook command.
- !326 - #720: Simplified the test demonstrating the difference in Python 2 and 3 behaviors.
- !327 - #720: Finalize embedded Python interpreter on exit from q.
- !331, !343 - #768: Improve C coverage

Improvement in the (internal) CI:

- !305, !309, !311, !321, !335, !347 - Multiple improvements in the CI.
- !319 - #770: Run doctests in tox.

PyQ 3.8.1

Released on 2016-06-21

- !292 - #744: Print guessed path of q executable when exec fails.
- !293, !294 - #748 Use VIRTUAL_ENV environment variable to guess QHOME.
- !301, !295 - #751: Update documentation.
- !296 - #750: Fall back on 32-bit version of q if 64-bit version does not run.
- !298, !299, !300, !303 - #753: CI Improvements.
- !302 - #755: Use preserveEnumerations=1 option to b9 instead of -1.

PyQ 3.8

Released on 2016-04-26.

- !256 - #670: Enable 32-bit CI
- !258 - #717 Expose sd0 and sd1 in python.
- !259 - #718 Added a test running "q test.p".
- !261 - Use Python 3.4.3 in CI
- !272, !273 - #731 Added Python 3.5.0 test environment and other CI improvements.
- !263 - #718 More p) tests
- !264 - #709 Redirect stderr and stdout to notebook
- !271 - #729 Conversion of lists of long integers to q.
- !274 - #728 Don't corrupt existing QHOME while running tox.
- !275 - #733 Don't add second soabi for Python 3.5.
- !276 - #734: Added support for enums in memoryview.
- !277 - #736: Implemented format() for more scalar types.
- !278 - #737 Misleading error message from the list of floats conversion.
- !279, !280 - #738 CI improvements
- !281 - #611: Updated k.h as of 2016.02.18
- !286, !288, !289, !290 - #742 PyQ Documentation
- !287 - #745: Automatically generate version.py for PyQ during setup.

PyQ 3.7.2

Released on 2015-07-28.

- !270 - #726 Reuse dict converter for OrderedDict.
- !267 - #724 and #723 numpy <> q conversion fixes.
- !266 - #725 Use 001..002 to bracket ANSI escapes.
- !265 - #721 Made slicing work properly with associations (dictionaries) and keyed tables.

- !260 - #719 Backport python 3 bug fixes.
- CI Improvements (!257, !262, !269, !268).

PyQ 3.7.1

Released on 2015-02-12.

- !244 - #701 Fixed using q datetime (z) objects in format().
- !246 - Removed pytest-pyq code. pytest-pyq is now separate package.
- !247 - #709 IPython q-magic improvements
- !248 - #673 Implemented unicode to q symbol conversion in python 2.x.
- !249, !252 - #691 Improved test coverage
- !250, !251 - #695 Use Tox as test-runner
- !253 - #715 Fixed table size computation in getitem.
- !255 - #691 Remove redundant code in slice implementation

PyQ 3.7

Released on 2015-01-15.

- !222 - #581 Implements conversion of record arrays.
- !223 - #680 Fixed int32 conversion bug.
- !224 - #681 Fixed datetime bug - freed memory access.
- !225 - Added support for numpy.int8 conversion.
- !226 - #644 Fixed descriptor protocol.
- !227 - #663 Fixed nil repr (again).
- !228, !233, !237, !239 - #687 Updates to documentation in preparation to public release.
- !229 - #690 Use only major kx version in _k module name.
- !230 - #691 Added tests, fixed date/time list conversion.
- !232 - #693 Implement pyq.magic.
- !234 - #694 Use single source for python 2 and 3. (No 2to3.)
- !235 - #674 Added support for nested lists.
- !236 - #678 Fixed compiler warnings.
- !238 - #657 Make numpy optional.
- !240 - #674 Added support for nested tuples.
- !241 - #696 Implemented slicing of K objects.
- !242 - #699 int and float of non-scalar will raise TypeError.
- !243 - #697 Fixed a datetime bug.

PyQ 3.6.2

Released on 2014-12-23.

- !198 - #654 Restore python 3 compatibility
- !211 - #667 Added pyq.c into MANIFEST
- !213 - #669 Fixed a crash on Mac
- !214 - #590 Implemented numpy date (M8) to q conversion
- !215, !216 - #590 Implemented support for Y, M, W, and D date units
- !217, !218, !220, !221 - #666 Multiple CI improvements
- !219 - #676 Implemented numpy.timedelta64 to q conversion

PyQ 3.6.1

Released on 2014-11-06.

- !206 - #663 Fixed nil repr
- !207 - CI should use cached version of packages
- !208 - #665 Allow K objects to be written into ipython zmq.iostream
- !209 - Show python code coverage in CI
- !210 - #666: Extract C and Python coverage to print in the bottom of the CI run
- !212 - Bump version to 3.6.1b1

PyQ 3.6.0

Released on 2014-10-23.

- !189 - #647 Fix pyq.q() prompt
- !190 - CI should use Python 2.7.8
- !191 - #648 Boolean from empty symbol should be False
- !192 - #634: Moved time converter to C and removed unused converters
- !193 - #652 Added __long__ method to K type.
- !194 - #653 Allow K integer scalars to be used as indices
- !195, !197 - #651 Format for scalar types D, M, T, U, and V.
- !196 - #611 Updated k.h to 2014.09.11
- !199 - #656 Iteration over K scalars will now raise TypeError.
- !200 - #655 Added support for Python 3 in CI
- !202 - #571 Added support for uninstalling Q components
- !203 - #633 Improve test coverage
- !204 - #633 Added boundary and None checks in ja

PyQ 3.5.2

Released on 2014-07-03.

- !184, !186 - #639 taskset support. Use CPUS variable to assign CPU affinity.
- !187 - #641 color prompt
- !185 - #640 Restore minimal support for old buffer protocol

PyQ 3.5.1

Released on 2014-06-27.

- !177, !178 – #631 pyq is binary executable, not script and can be used in hashbang.
- !179 – #633 Added memoryview tests.
- !181 – #636 Moved extension module into pyq package.
- !182 – #633 Removed old buffer protocol support.
- !183 - #638 Calling q() with no arguments produces an emulation of q) prompt

PyQ 3.5.0

Released on 2014-06-20.

- !164 – #611 Updated k.h
- !165 – #614 Expose jv
- !166 – #580 Show with output=str will return string
- !167 – #627 Fixed p language
- !168 – Fix for pip, PyCharm and OS X
- !169 – #629 python.py script was renamed to pyq
- !170 – #632 jv reference leak
- !171 – #633 C code review
- !172 – #634 k new
- !173 – #612 Generate C code coverage for CI
- !174, !175 – #633 test coverage
- !176 – #635 Disable strict aliasing

PyQ 3.4.5

Released on 2014-05-27.

- 614: Expose dj and ktj
- 620: Empty table should be falsy
- 622: Convert datetime to “p”, not “z”

PyQ 3.4.4

Released on 2014-05-23.

- python.q returns correct exit code

PyQ 3.4.3

Released on 2014-04-11.

- 617: Dict Conversion
- 619: Len Keyed Table

PyQ 3.4.2

Released on 2014-04-11.

- 589: Symbol array roundtripping
- 592: Properly register py.path.local
- 594: Support passing additional values to select/update/exec methods.
- 595: Implement pytest_pyq plugin
- 596: Implement python dict converter
- 601: Add support for ^ (fill) operator
- 602: Fix r-ops for non-commutative operations.
- 603: Fix unary + and implement unary ~
- 604: Make all q methods accessible from pyq as attributes
- 609: Updated k.h to the latest kx version
- NUC: Only true division is supported. Use "from __future__ import division" in python 2.x.

PyQ 3.4.1

Released on 2014-03-14.

- Add support for char arrays #588
- PyQ can now be properly installed with pip -r requirements.txt #572

PyQ 3.4

Released on 2014-03-07.

- Issues fixed: #582, #583, #584, #586
- Support dictionary/namespace access by .key
- Support ma.array(x) explicit conversion
- Add support for comparison of q scalars

PyQ 3.3

Released on 2014-02-05.

- Issues fixed: #574, #575, #576, #577, #578

PyQ 3.2

Released on 2013-12-24.

- Issues fixed: #556, #559, #560, #561, #562, #564, #565, #566, #569, #570, #573
- **NEW: wrapper for python.q to use it under PyCharm** Note: You will need to create symlink from python to python.py in order for this to work, i.e.: ln -s bin/python.py bin/python
- Support to use 32-bit Q under 64-bit OS X

PyQ 3.2.0 beta

- Convert int to KI if KXVER < 3, KJ otherwise
- In Python 2.x convert long to KJ for any KXVER

PyQ 3.1.0

Released on 2012-08-25.

- support Python 3.2
- release pyq-3.1.0 as a source archive

2012-08-10

- basic guid support

PyQ 3.0.1

Released on 2012-08-09.

- support both q 2.x and 3.x
- better setup.py
- release pyq-3.0.1 as a source archive

2009-10-23

- NUC: k3i
- K(None) => k("::")
- K(timedelta) => timespan

2009-01-02

- Use k(0, ..) instead of dot() and aN() to improve compatibility
- Default to python 2.6
- Improvements to q script.p
- NUC: extra info on q errors

2007-03-30

implemented K._ja

0.3

- Added support for arrays of strings

0.2

- Implemented iterator protocol.

PyQ General License

PyQ is distributed as open source software and is free for the users of the 32bit edition of kdb+ under the terms of the *free 32-bit license*.

For more information, see the following sections.

Copyright

Free 32-bit license

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

0. This software or its derivatives must be run on a free 32-bit version of kdb+ subject to the DOWNLOAD KDB+ SOFTWARE LICENSE AGREEMENT.
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgment: This product includes software developed by the Enlightenment Research, LLC.

4. Neither the name of the Enlightenment Research, LLC nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY ENLIGHTENMENT RESEARCH, LLC ''AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL ENLIGHTENMENT RESEARCH, LLC BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

64-bit license

For a 64-bit license and support, please contact "PyQ License" at pyq-lic@enlnt.com.

CHAPTER 2

Navigation

- genindex
- search

Index

K

L

M

N

O

P

Q

R

S

T

U

V

W

X