QnD Package Documentation

Release 1.0.0

David Munro

Jul 11, 2023

Contents

genindex	modindex	search

The qnd module (quick and dirty) provides a frontend for reading and writing self-describing binary files. Backends exist for HDF5 (via h5py), netCDF3, and PDB (via pure python code contained in QnD) file formats. netCDF4 is not specifically supported, but since it's based on HDF5 such files can be treated as HDF5. Adding backends is not very difficult; the interface is well-defined and relatively small.

QnD is free and open-source, and hosted on LLNL's public github.

This manual describes the design philosophy behind this user interface; in a nutshell, the idea is to keep things as simple as possible, but no simpler (as Einstein said). The problem we are setting out to solve is to store collections of scientific data, which means for the most part arrays of numbers.

CHAPTER 1

Contents

1.1 User Interface for Binary Files

In terms of the scipy environment, qnd addresses the storage and retrieval of numpy ndarrays, excepting arrays with the general python object data type (dtype.kind 'O'). In scipy programs, these arrays may be attributes of objects surrounded by methods and other non-data, but we assume that the programmer provides some means of initializing all such objects given just the (numerical) ndarrays at their heart.

The python language provides two kinds of collections which qnd directly supports: the dict (a collection of named objects) and the list (a sequence of heterogeneous objects). We presume that a programmer will provide a way to map a nested tree (no loops) of dicts and lists with ndarray leaves (and dict keys which are strings) to and from whatever objects their program requires. The dict and list are precisely the collections provided by the simple and popular JSON data interchange format. Thus, in order to use the qnd storage interface, we are essentially asking the programmer support a portable organization of the program data.

Note the contrast to the goal of the python pickle module; we acknowledge that some extra design and maintenance work may be required to support such a mapping. Often the additional effort pays off in a simplified overall design.

1.1.1 Basic Usage

The first step is to obtain a file handle, say *f*, by opening a file. The open function belongs to the particular backend, called `openXXX`, and defined in a backend module `XXXf`. For example:

```
from qnd.h5f import openh5
f = openh5('filename.h5', 'r+')
```

The qnd mode choices are the same for all backends (copied from the excellent h5py module mode sematics):

- 'r' opens the file read-only
- 'w' creates a new file, clobbering any existing file
- 'a' opens the file read-write, creating it if it did not exist
- 'r+' opens the file read-write, raising an error if it did not exist

- 'w-' creates a new file, raising an error if it exists beforehand

These mode flags are not semantically identical to the python `open` function: The 'w-' is not recognized by `open` at all, and the `open` 'a' guarantees that any existing file bytes will not be modified, while the qnd 'a' merely means read-write (but has the same semantics as 'a' in terms of file existence and creation). Furthermore, qnd files are always readable, even if opened in one of the 'w' modes.

Python syntax has two operators for extracting named members from a compound object: The dot operator extracts an *attribute* from an object, and square brackets extract an *item* from a dict object (or other mapping). Qnd file handles support both. The dot syntax `f.var` is best when you know the name 'var' at the time you write the expression, while the square bracket syntax `f[expression]` is best when the name is the result of an expression or value of a variable.

The dot operator is overloaded, since it is also used for method attributes like `f.close()`. Thus the qnd file handle *f* really behaves like a dict for the most part, with its support for the dot syntax mere sugar to improve code legibility and, at least as importantly, ease of typing in interactive usage. If there were a variable named 'close' in *f* (who knows where *f* came from), you could always access it as `f['close']`. However, qnd provides a quick and dirty option for using the dot operator even in these cases: it will remove a single trailing underscore, so that `f.close_` refers to the variable `'close'`, not `'close_'`. (`f.close__` would refer to `'close_'`.) This idiom is suggested by the PEP8 python style guide, and you would also need it to escape python keywords, like `f.yield_` to refer to `f['yield']`.

The bottom line is, you use a qnd file handle *f* as if it were a python dict, but you are also free to treat items in *f* as if they were attributes of this dict object:

```
x = f.x           # read variable "x" from f, same as f['x']
f.x = expression  # declare and write "x" to f
f.update(x=expr1, y=expr2, ...)  # declare and write several variables
# update also accepts non-keyword dicts and lists of (name, value)
x = f.get('x', xdefault)  # same as get from dict
varnames = list(f)        # preferred over f.keys(), as for any dict
nvars = len(f)
if name in f: do_something
for name in f: do_something
for name, value in f.items(): do_something
```

In addition to the dict-like *update*, *get*, *keys*, and *items* methods, qnd files also have a number of non-dict methods and behaviors:

```
f.close()
f.flush()  # like close then reopen
with openh5('myfile.h5', 'a') as f:
    write_something(f)  # closing f upon exit from with suite
f.auto(0)        # turn off (or on) auto-read mode
f.recording(1)   # turn on (or off) recording mode
f.goto(time=t)   # set to previously recorded record
with f.push():
    do_something(f)  # temporarily change auto, recording, goto state
```

The *recording* and *goto* modes are the subject of the next section; we conclude this section by discussing *auto* mode. You may have noticed that `f.x` or `f['x']` immediately read the variable from the file, giving you no opportunity to query its data type or shape, which you might well want to do without incurring the overhead of the actual read, especially if you know it is a very large array. We can get the names of all stored variables with `list(f)`, but how do we find out what each one looks like without reading it?

The answer is that a qnd file *f* can be placed into a mode in which variable references do not trigger an automatic read operation, by invoking `f.auto(0)`. You can also request this mode using the `auto=0` keyword when you open the file. (The default is `auto=1`.) With autoread mode off, getting an item returns a qnd leaf object, which is like a mini-file handle you can use to query, read, or write only that specific variable. It has properties similar to an ndarray:

```
f.auto(0)
xhandle = f.x  # or f['x']
dtype, shape = xhandle.dtype, xhandle.shape  # also size and ndim
xhandle = f(0, 'x')  # return handle to x independent of auto mode
x = xhandle[:]  # read x if x is not scalar
x = xhandle[()]  # read x no matter what
xhandle[()] = expression  # write x no matter what
x = xhandle()  # shorthand for xhandle[()]
xhandle(expression)  # shorthand for xhandle[()] = expression
xpart = xhandle[index_expressions]  # read part of x
xhandle[index_expressions] = xpart  # write part of x
```

Notice that *xhandle* inherits the obscure indexing behavior of ndarray scalars, for which `x[:]` raises an error. However, *xhandle* provides a non-ndarray operation to compensate – calling a qnd handle as a function always reads the whole thing, whether or not it has any dimensions.

Although the qnd leaf handles can be used for partial read and write operations, if that is all you want to do, you can simply combine the partial index expressions into a single square bracket:

```
xpart = f['x', index_expressions]
f['x', index_expressions] = xpart
```

These work no matter how the autoread mode is set, but there is no equivalent using the dot syntax: Although `f.x[index_expressions]` produces the same final result, it reads all of *x* before applying *index_expressions* to the resulting large ndarray.

(Note that qnd only reads or writes the largest contiguous block of leading indices specified by *index_expressions*; it only reduces the intermediate memory footprint when the leading indices are scalar or small slices of *x*.)

Finally, sometimes you need to declare a variable without writing it. To do this in qnd, make its value a dtype or a (dtype, shape) tuple:

```
f.x = float  # declare x to be a scalar dtype(float), that is f8
f.y = yy.dtype, yy.shape  # declare y with type and shape of yy
f.z = bool, yy.shape  # declare z to be boolean with same shape as yy
```

Such a declaration reserves space for the array in the file, but it is your responsibility to fill it with sensible values with one later write or several partial writes.

1.1.2 Recording History

Setting an item with `f.x = value` or `f['x'] = value` both declares the variable and writes its value. If you later write it a second time with `f.x = value2`, by default this overwrites the orginal value you wrote. Sometimes, however, you need to record the history of a variable which is changing as a simulation progresses. The idea behind recording mode is to make the second assignment store the new *value2* in addition to the original *value*, so by repeatedly assigning values to *x* you can store as many versions of its changing values as you like.

The HDF5, netCDF, and PDB file formats all support this capability by allowing the leading dimension of a variable to be "unlimited". But in qnd, you can suppress this fictitious leading dimension by using the *recording* mode to write such variables, and the *goto* mode to read them:

```
f = openh5('myfile.h5', 'w')
f.x = xa  # x is not a record variable.
f.recording(1)  # Put f in recoding mode; new variables are recorded.
f.time = t0  # Time is a record variable with t0 for its first record.
f.y = y0  # y is a record variable with y0 for its first value.
```

(continues on next page)

(continued from previous page)

```
f.x = xb  # x remains a non-record variable, xb overwrites xa
f.time = t1  # Write a second record of time with value t1.
f.y = y1  # Write a second record of y with value y1.
f.close()

f = openh5('myfile.h5', 'r')
# Initially, goto mode is off (None), and reading a record variable...
times = f.time[:]  # ...returns a list (not array) of all of its records.
# Use goto to set a "current record" index for all record variables:
f.goto(0)  # first record
t0 = f.time
y0 = f.y
xb = f.x  # non-record variables ignore current record
with f.push():  # current record restored on exit from with suite
    f.goto(-1)  # go to last record, record<0 acts like any other index
    yN = f.y
# You may use any scalar record variable as a keyword to jump to the
# record nearest the specified value of that variable (assuming it is
# monotonic):
f.goto(time=1.2)  # set to record where f.time nearest 1.2
y12 = f.y
for record in f.gotoit():  # iterate over all records
    # gotoit() causes implicit f.goto(record) before each pass
    do_something(f)
f.goto(None)  # Turn off goto mode.
ylist = f.y  # list of y arrays at every record
```

The qnd interface, unlike the existing backend file formats, also supports the case of record variables whose shape changes from one record to the next. To use this feature, set the recording mode to 2 instead of to 1:

```
f.recording(2)
f.x = zeros((nx, ny))  # First x record has shape (nx, ny).
f.x = zeros((nx+5, ny-2))  # Second x record has shape (nx+5, ny-2).
f.goto(None)
xlist = f.x  # list of x arrays at every record
```

This possibility explains why `f.recordvar` returns a list of values at every record, rather than an array with an extra leading dimension (as in the fiction employed for the existing file formats).

1.1.3 Groups and lists of variables

The qnd file handle class is *QGroup*; specifically it is the "root group" of the file. But a QGroup may contain subgroups, just as a python dict may contain other dicts. To define a subgroup, simply assign a dict instead of an array-like value to an item:

```
f.g = {}  # declare an empty subgroup g
f.g.update(x=expr1, y=expr2)  # all the methods of f work with g
g = f.g  # g is a QGroup, a subgroup of f
y = g.y  # or g['y']
g.auto(0)  # initially g inherits autoread and other modes from f
root = g.root()  # returns root QGroup, root is f here
if f is f.root(): task_if_f_is_root_group()
f['g/x']  # same as f.g.x
f['/g/x']  # same as f.root().g.x
```

Although a subgroup initially inherits its autoread, recording, and goto modes from its parent, thereafter the modes of *g* are independent of the modes of *f*. In a *gotoit* loop, the record number in the iterator will be necessary to explicitly keep subgroups synchronized:

```
g = f.g
for record in f.gotoit():
    g.goto(record)
    do_something(f, g)
```

Because of the the fact that a *QGroup* looks like a dict, `dict(f)` will read every variable in *f*. By analogy with the qnd leaf handles, `f()` also reads every item in *f* into a dict, with one twist: Instead of an ordinary dict, `f()` results in a dict subclass called an *ADict*, which permits access to the dict items as attributes according to the same rules as for a *QGroup*. If you want to convert your own *dict* objects into *Adict* objects, you can use the *redict* function in the `qnd.adict` module. That module also contains a generic mix-in class *ItemsAreAttrs* which you can use as a base class for your own mapping classes. (Although be sure you read the comment in the *__getattr__* method before you attempt this, as it can make your code difficult to debug.)

Note that `f()` respects the autoread and goto modes. Thus if `auto=0`, you nothing will be read from the file and the returned dict will contain qnd leaf handles (*QLeaf* objects) rather than variable values. When `auto=1`, the dict item corresponding to any subgroup will be a *QGroup* object. If you want to recursively read all subgroups, set `auto=2`, which causes subgroups to be read automatically. (Note that since `g = f.g` produces an *ADict* in that case rather than a *QGroup*, `auto=2` can never be inherited.)

In addition to *QGroup* (a dict with str keys) and *QLeaf* (an ndarray), the qnd interface provides a third item type, *QList*, which stores a python heterogeneous list. A *QList* is a way to store a sequence of objects anonymously, so that you can reference them simply by a sequence number instead of by a name. If you find yourself inventing sequences of names like 'var00', 'var01', var02', and so on, to store in a *QGroup*, you want to use a *QList* instead:

```
f.var = list  # (the builtin list type) declares empty list var
var = f.var  # the QList object, assuming f.goto(None)
var.append(value0)  # QList has list-like append and extend methods
var.append(value1)
var.extend([value0, value1, ...])
value1 = var[1]  # second item of var, negative index, slices work
var[1] = newvalue1  # overwrite value1
nitems = len(var)
var.auto(0)  # QList initially inherits its parent's autoread mode
```

Although *QList* has an autoread mode like a *QGroup*, it does not have either a recording mode or a goto mode. In fact, a record variable is implemented as a *QList*, so the recording and goto modes in the parent group will influence how the list presents itself:

```
f.goto(1)
value1 = f.var  # In goto mode, f.var means f.var[current_record].
```

The ability to store aribtrary str-keyed dict and list trees whose leaves are ndarrays (or None) gives qnd the ability to support pretty much arbitrary python objects. In particular, anything which can be reduced to JSON format can be stored.

1.1.4 Other attributes

The HDF5 and netCDF file formats support variable attributes beyond name, type, and shape. These attribute metadata are generally not useful outside a very narrow software suite for which they were designed, but may provide helpful documentation when first opening a category of file. Therefore, qnd supports variable attributes for backend formats which support them. In qnd, all attributes belong to the *QGroup* of the parent. Thus, *QList* elements may not have attributes (which is irrelevant since neither HDF5 nor netCDF has native support for list objects):

```
fattrs = f.attrs()
attrs = fattrs.x  # or fattrs['x'], attributes of f.x
attrs = fattrs._  # or fattrs[''], attributes of f itself
value = attrs.aname  # or attrs['aname'] value of attribute or None
attrs.aname = value  # declare and set attribute
attrs.aname = dtype, shape, value  # convert value to dtype and shape
anames = list(fattrs.x)  # names of attributes of f.x
if aname in fattrs.x: do_something
for aname in fattrs.x: do_something
for aname, avalue in fattrs.x.items(): do_something
```

Attribute values may not be dict or non-array-like lists. Also, the attribute names 'dtype', 'shape', 'size', 'ndim', and 'sshape' will always return the corresponding properties of the item, even though they are not stored as variable attributes and are not actually present in the *attrs* mapping objects.

genindex	modindex	search

1.2 qnd.adict module

An items-as-attributes dict.

class qnd.adict.**ADict**(**args*, ***kwargs*)

Bases: *qnd.adict.ItemsAreAttrs*, dict

Subclass of dict permitting access to items as if they were attributes.

For a ADict ad, `ad.x` is equivalent to `ad['x']` for getting, setting, or deleting items. The exceptions are dict method names, like *keys* or *items*, syntactically illegal names, like *class* or *yield*, and any name beginning with __.

Additionally, as a work around for some of these exceptions, ADict will remove a single trailing underscore from an attribute name, so `ad.x_` is also equivalent to `ad['x']`, and you need `ad.x__` to get `ad['x_']` (a convention inspired by the similar PEP8 recommendation for syntatically illegal variable names). The trailing underscore removal does not apply to names beginning with __.

The trailing underscore removal convention applies to keywords passed to the constructor or to the *update* method as well.

Use subscript syntax when a variable or expression holds an item name; use attribute syntax when you know the item name at parse time:

```
ad[variable] = value  # value of variable is the item name
ad.fixed = value  # 'fixed' is the item name
value = ad.get('fixed', default)  # except to avoid KeyError
```

See also:

redict recursively toggle between dict and ADict

ItemsAreAttrs mixin base class to provide this for any class

class qnd.adict.**ItemsAreAttrs**

Bases: object

Mix-in class for QArray, QGroup, or QList, and also ADict.

update(**args*, ***kwargs*)
Multiple __setitem__ from positional arguments or keywords.

qnd.adict.**redict**(*d*, *cls=None*)
Recursively convert a nested dict to a nested ADict and vice versa.

Parameters

- **d** (*dict or ADict instance*) – A dict, possibly nested, to be converted.
- **cls** (*dict or subclass of dict, optional*) – The dict-like cls to recursively convert *d* and any sub-dicts into. By default, if *d* is a *ADict*, *cls* is *dict*, otherwise *cls* is *ADict*, so repeated calls to *redict* toggle between *dict* and *ADict*.

Returns **dnew** – A copy of *d* whose class is *cls*. Any items which are dict instances are similarly copied to be *cls* instances. Non-dict items are not copied unless assignment makes copies.

Return type dict or *ADict*

genindex	modindex	search

1.3 qnd.frontend module

Quick and Dirty, a high level file access wrapper.

The QnD interface looks like this:

```
f = format_specific_open(filename, mode)  # e.g. openh5
var = f.varname  # read var
var = f['varname']
var = f.get('varname', default)

f.var = var_value  # declare and write var
f['var'] = something
f.var = dtype, shape  # declare var without writing
f.update({...vars...}, another_var=value, ...)
f.grpname = {...vars...}  # declare a subgroup and some members

if name in f: do_something
varnames = list(f)
for name in f: do_something
for name, var in f.items(): do_something

g = f.grpname
f = g.root()  # Use the root method to get the top-level QGroup.
f.close()  # important if you have written to f
```

Generally, a QnD QGroup like *f* in the example behaves like a dict. However, you may also reference variables or subgroups as if they were attributes. Use attributes to access variables when you know the variable name. In short, use square brackets when the variable name is the value of an expression. (QnD will remove a single trailing underscore from any attribute reference, so you can use `f.yield_` for `f['yield']` or `f.items_` for `f['items']`.) The adict module has an *ADict* class and a *redict* function to produce ordinary in-memory dict objects with their items accessible as attributes with the same rules. You can read a whole file (or a whole subgroup) like this:

```
ff = f(2)
```

The optional *2* argument is the auto-read mode flag. By default, the auto-read mode flag is set to 1, which causes `f.varname` to read an array variable and return its value, but to simply return a QGroup object (like *f*) if the name refers to a subgroup. When the *auto* flag equals *2*, any subgroups are read recursively, and their values become ADict instances. (QnD also supports QList variables, and `auto=2` mode returns those as python list instances.)

The `items()` method also accepts an optional *auto* argument to temporarily change auto-read mode used for the iteration.

You can turn auto-read mode off by setting the *auto* flag to *0*. In this mode, referencing a variable returns a QLeaf instance without reading it. This enables you to query a variable without reading it. You can also do that by retrieving the attributes object:

```
with f.push():  # Use f as a context manager to temporarily change modes.
    f.auto(0)  # Turn off auto-read mode.
    v = f.varname
value = v()  # Read a QLeaf by calling it...
value = v[:]  # ...or by indexing it.
v(value)  # Write a QLeaf by calling it with an argument...
v[:] = value  # ...or by setting a slice.
v.dtype, v.shape, v.size, v.ndim  # properties of the QLeaf v
# An alternate method which pays no attention to auto mode:
va = f.attrs.varname  # Get attributes of varname.
va.dtype, va.shape, va.size, va.ndim  # Built-in pseudo-attributes.
# You can use va to get or set real attributes of varname as well:
units = va.units  # retrieve units attribute
va.centering = 1  # set centering attribute
```

When you call a QGroup like *f* as a function, you may also pass it a list of variable names to read only that subset of variables. With auto-read mode turned off, this results in a sort of "casual subgroup":

```
g = f(0, 'vname1', 'vname2', ...)
h = f(1, 'vname1', 'vname2', ...)
ff = f(2, 'vname1', 'vname2', ...)
```

Here, g is an ADict containing QLeaf and QGroup objects, with nothing at all read from the file, while h is and ADict containing ndarray and QGroup objects, while ff is an ADict containing ndarray and ADict objects, with no references at all to *f*.

If you want to use *f* as a context manager in the manner of other python file handles, so that the file is closed when you exit the with statement, just do it:

```
with openh5(filename, "a") as f:
    do_something(f)
# f has been properly flushed and closed on exit from the with.
```

QnD also supports old netCDF style UNLIMITED dimensions, and their equivalents in HDF5. Unlike the netCDF or HDF5 interface, in QnD the first (slowest varying) dimension of these arrays maps to a python list, so we regard the entire collected variable as a list of ndarrays. The netCDF record number is the index into the list, while any faster varying dimensions are real ndarray dimensions. This subtle difference in approach is more consistent with the way these variables are stored, and also generalizes to the fairly common case that the array dimensions – often mesh dimensions – change from one record to the next.

To write records using QnD, turn on "recording mode":

```
f.recording(1)  # 0 for off, 2 for generalized records
f.time = 0.
```

(continues on next page)

(continued from previous page)

```
f.x = x = arange(10)
f.time = 0.5
f.x = x**2
```

Ordinarily, when you set the value of `f.time` or `f.x`, any previous value will be overwritten. But in recording mode, each time you write a variable, you create a new record, saving the new value without overwriting the previous value. If you want all record variables to have the same number of records, you need to be sure you write them each the same number of times. One way to do that is to use the update function rather than setting them one at a time:

```
record = ADict()
record.time, record.x = 0., arange(10)
f.recording(1)
f.update(record)
record.time, record.x = 0.5, record.x**2
f.update(record)
```

You cannot change a variable from not having records to having records (or from recording mode 1 to recording mode 2); the recording mode in force when a variable was first declared determines if and how all future write operations behave.

Reading back record variables introduces "goto mode". Initially, goto mode is off or None, so that reading a record variable gets the whole collection of values as a QList, or as an ordinary python list if auto mode is on:

```
f.goto(None)  # explicitly turn off goto mode
f.auto(2)
times = f.time  # python list of f.time values
xs = f.x  # python list of f.x arrays
f.auto(0)
time = f.time  # QList for the collection of time values
nrecords = len(time)
```

On the other hand, with goto mode turned on, the fact that *time* and *x* are record variables disappears, so that your view of `f.time` and `f.x` matches what it was when you recorded them. You use the goto function to set the record:

```
f.goto(0)  # first record is 0, like any python list
t = f.time  # == 0.
f.goto(1)  # set to second record
t = f.time  # == 0.5
x = f.x  # == arange(10)**2
f.goto(-1)  # final record, negative index works like any python list
# You can also pass a keyword to goto, which can be the name of any
# scalar record variable, to go to the record nearest that value.
f.goto(time=0.1)  # will select record 0 here
current_record = f.goto()  # goto() returns current record number

for r in f.gotoit(): do_something  # f.goto(r) is set automatically
```

Note the `gotoit()` method returns an iterator over all records, yielding the record number for each pass, and setting the goto record for each pass automatically. You can use `f.push()` in a with statement to temporarily move to a different record.

If you set the recording mode to *2*, the record variables need not have the same shape or same type from one record to the next (indeed, they can be a subgroup on one record and an array on another). This cannot be represented as an UNLIMITED array dimension in an HDF5 or netCDF file, so the QList variable in QnD will become an HDF5 group in this case, where variable names in the group are _0, _1, _2, and so on for QList element 0, 1, 2, and so on (plus a hidden element _ which identifies this group as a list when it is empty). You can create a QList of this general type without using recording or goto mode at all:

```
f.recording(0)  # Turn off recording and goto modes.
f.goto(None)
f.varname = list  # Make an empty QList
ql = f.varname
ql.append(value0)
ql.extend([value1, value2, ...])
var = ql[1]  # retrieves value1
nelements = len(ql)  # current number of elements (also works for QGroup)
ql.auto(0)  # a QList has auto mode just like a QGroup
for var in ql: do_something  # var depends on ql auto mode setting
```

---

class qnd.frontend.**QAttributes**(*parent*, *vname=None*)

Bases: *qnd.adict.ItemsAreAttrs*

Attributes for a QGroup and its members.

Usage:

```
qa = qgroup.attrs()
qa0 = qa.vname  # for variables in this group, or qa['vname']
qa1 = qa._  # or qa[''] for attributes of this group
value = qa0.aname  # or qa0['aname'], None if no such attribute
qa0.aname = value  # or qa0['aname'] = value
qa0.aname = dtype, shape, value
if 'aname' in qa0: do_something
for aname in qa0: do_something
for aname, value in qa0.items(): do_something
```

class qnd.frontend.**QGroup**(*item=None*, *state=None*, *auto=None*, *recording=None*, *goto=None*)

Bases: *qnd.adict.ItemsAreAttrs*

Group of subgroups, lists, and ndarrays.

You reference QGroup items by name, either as `qg['name']` like a dict item, or equivalently as `qg.name` like an object attribute. Use `[]` when the item name is an expression or the contents of a variable; use `.` when you know the name of the item. You can use `[]` or `.` to both get and set items in the QGroup. To read the entire group into a ADict, call it like a function, `qg()`; you may supply a list of names to read only a subset of items. A QGroup acts like a dict in many ways:

```
if 'name' in qg: do_something
for name in qg: do_something
item_names = list(qg)  # qg.keys() exists but is never necessary
for name, item in qg.items(): do_something
qg.update({name0: val0, ...}, [(name1, val1), ...], name2=val2, ...)
value = qg.get('name', default)
```

A QGroup has several possible states or modes:

1. Recording mode, turned on by `qg.recording(1)` and off by `qg.recording(0)`, affects what happens when you set group items. With recording mode off, setting an item to an array creates the item as an array if its name has not been used, or otherwise writes its new value, requiring it be compatible with the dtype and shape of the previous declaration. With recording mode on, setting an item for the first time creates a QList and sets its first element to the given value, and subsequently setting that item appends the given value to the existing QList. There is also a recording mode `qg.recording(2)` in which subsequent values need not match the dtype or shape of the first item. You may not switch recording modes for a given item; the mode in effect when an item is first created governs the behavior of that item.

2. Goto mode, in which you set a current record with `qg.goto(rec)`. Any item you retrieve or query which is a QList retrieves or queries the element with 0-origin index `rec` instead of the whole QList. You turn off goto mode with `qg.goto(None)`. There is also a `qg.gotoit()` function which returns an iterator over all the records (generally the longest QList in `qg`).

3. Auto mode, turned on by `qg.auto(1)` and off by `qg.auto(0)`, in which getting any item reads and returns its value, rather than a QLeaf object. There is also a `qg.auto(2)` mode in which the auto-read feature applies to any QGroup or QList (if goto mode is off) items recursively.

A QGroup has *push* and *drop* methods which can be used to save and restore all its modes. The *drop* method is called implicitly upon exit from a with statement, so you can use the QGroup as a context manager:

```
with openh5('myfile.h5', 'a') as qg:
    do_something(qg)
    with qg.push():
        qg.goto(rec)
        do_something_else(qg)
    # qg restored to goto mode state before with.
    do_even_more(qg)
# qg flushed and closed upon exit from with clause that has no
# no corresponding push
```

islist

isleaf
Always 0.

isgroup
Always 1.

dtype
Always `dict`, the builtin python type.

shape

ndim

size

sshape
Always None.

attrs()
Return attribute tree for variables in this group.

auto(*recurse*)
Set the auto-read mode for this QGroup.

In auto-read mode, getting an item returns its value, rather than a QLeaf. If the item is a QGroup or QList, that is returned if the *recurse* value is 1, whereas if *recurse* is 2, the QGroup or QList variables will be read recursively. Setting *recurse* to 0 turns off auto-read mode entirely.

Note that you can temporarily set auto mode using a with clause.

close()
Close associated file.

drop(*nlevels=None*, *close=False*)
Restore previous recording, goto, and auto mode settings.

Default `drop()` drops one pushed state, `drop(n)` drops n, `drop('all')` drops all pushed states. By default, *drop* is a no-op if no pushed states to drop, `drop(close=1)` closes the file if no pushed states to drop, which is called implicitly on exit from a with suite.

dtype
alias of builtins.dict

flush()
Flush associated file.

get(*key*, *default=None*)
like dict.get method

goto(*record=<object object>*, ***kwargs*)
Set the current record for this QGroup, or turn off goto mode.

Pass *record* of None to turn off goto mode, so that QList variables appear as the whole QList. Setting an integer *record* makes any QList variable appear to be the specified single element. A *record* value may be negative, with the usual python interpretation for a negative sequence index. If different QList variables have different lengths, the current *record* may be out of range for some variables but not for others. (Hence using goto mode may be confusing in such situations.)

Note that you can temporarily set goto mode using a with clause.

This *goto* method also accepts a keyword argument instead of a *record* number. The keyword name must match the name of a QList variable in this QGroup, whose vaules are scalars. This will set *record* to the record where that variable is nearest the keyword value. Thus, `goto(time=t)` selects the record nearest *time* t.

As a special case, you can get the current record number by calling *goto* with neither a *record* nor a keyword:

```
current_record = qg.goto()
```

gotoit(*name=None*)
Iterate over goto records, yielding current record.

Optional *name* argument is the name of a *goto* method keyword, which may implicitly remove records corresponding to non-monotonic changes of that variable. If *name* is a decreasing variable, the record order will be reversed.

As a side effect, the current record of this QGroup will be set during each pass. If the loop completes, the original goto state will be restored, but breaking out of the loop will leave the goto record set.

items(*auto=None*)
like dict.items method (iteritems in python2)

push()
Push current recording, goto, and auto mode onto state stack.

recording(*flag*)
Change recording mode for this QGroup.

With recording mode off, writing to a variable overwrites that variable. With recording mode on, new variables are declared as a QList and subsequent write operations append a new element to this QList instead of overwriting any previously stored values. In netCDF parlance, variables declared in recording mode are record variables. Writing to a variable declared when recording mode was off will always overwrite it; once declared, you cannot convert a variable to a QList simply by turning on recording mode.

See goto mode for handling record variable read operations.

A *flag* value of 0 turns off recording mode. A *flag* of 1 turns on recording mode, utilizing a trailing UNLIMITED array dimension in netCDF or HDF5 parlance, which promises that all values written will have the same dtype and shape. A *flag* of 2 places no restrictions on the dtype or shape of the QList elements; such an unrestricted QList resembles an anonymous QGroup.

root()
Return root QGroup for this item.

class qnd.frontend.**QLeaf**(*item*)
Bases: object

An ndarray or None stored in a file.

You can read the data by calling the leaf instance `ql()`, or by indexing it `ql[:]`, which also provides a means for partial reads. A QLeaf has *dtype*, *shape*, *ndim*, and *size* properties with the same meanings as an ndarray (except None has all these properties equal None). Additionally, the *sshape* property may return a symbolic shape with optional strings in the tuple representing dimension names.

You can write data by calling `ql(value)`, or by setting a slice, which provides a means for partial writes.

isgroup

islist
Always 0.

isleaf
Always 1.

dtype
The numpy dtype of this ndarray, or None if this leaf is None. This is the dtype in memory, not necessarily as stored.

shape

ndim

size
The numpy ndarray properties, or None if this leaf is None.

sshape
A symbolic shape tuple, like shape except dimension lengths may be type str instead of int.

root()
Return root QGroup for this item.

class qnd.frontend.**QList**(*item=None*, *auto=0*)
Bases: object

List of subgroups, lists, and ndarrays.

You reference QList elements by index or slice, like ordinary list elements, including the python convention for negative index values. To read the entire list, call it like a function, `ql()`, which is equivalent to `ql[:]`. A QList has __iter__, append, and extend:

```
for element in ql: do_something
ql.append(value)
ql.extend(iterable)
```

In general, the elements of a QList are unrelated to one another; it's like an anonymous QGroup. However, a common use case is to represent a so-called UNLIMITED dimension in netCDF or HDF5. In this case, every element will have the same dtype and shape. The *islist* method returns 1 for this special restricted case, while it returns 2 for an unrestricted QList. Whether this makes any difference depends on the underlying file format. The QGroup *recording* and *goto* methods allow you to access QList items in the group transparently, as if they were individual elements at a current record or index.

isgroup

isleaf
Always 0.

islist
This is 1 if this QList is a record array declared in recording mode 1, and 2 if it was declared in any other way (including as a record array in recording mode 2).

dtype
Always `list`, the builtin python type.

shape

ndim

size

sshape
Always None.

append(*value*)
append a new element to this QList

auto(*recurse*)
Set auto read mode, analogous to QGroup.auto method.

dtype
alias of `builtins.list`

extend(*iterable*)
append multiple new elements to this QList

root()
Return root QGroup for this item.

class qnd.frontend.**QState**(*recording=0*, *goto=None*, *auto=0*)
Bases: `list`

State information for a QGroup.

class qnd.frontend.**QnDList**(*parent*, *empty=None*)
Bases: `object`

Implmentation of a low level QList type using QGroup.

A backend which has no direct support for QList objects can use this to produce a pseudo-list, which is a group with member names _ (None or a single signed or unsigned byte, value never read) and names _0, _1, _2, etc.

This implementation will handle both UNLIMITED index-style lists made with recording = 1 (that is group.declare with unlim flag) and general lists. If UNLIMITED dimensions are supported, pass the QnDLeaf to this constructor:

```
item = QnDList(QnDLeaf)  # if at least one record exists
item = QnDList(QnDLeaf, 1)  # if no records yet exist
```

Use the fromgroup constructor to check if a QnDGroup is a pseudo-list:

```
item = QnDList.fromgroup(QnDGroup)
```

genindex	modindex	search

1.4 qnd.generic module

Generic file or file family open.

class qnd.generic.**MultiFile**(*pattern*, *existing*, *future*, *mode*, ***kwargs*)
Bases: object

A binary file or family of binary files.

callbacks(*flusher*, *initializer*)
set callback function that flushes file metadata

close()
flush and close the current file

current_file()
Index of current file in family, argument to open method.

declared(*addr*, *dtype*, *nitems*)
declare that array has been declared, maybe update next_address

filename(*n=None*)
current or n-th existing filename in family

flush()
flush metadata and ordinary file buffers

next_address(*both=False*, *newfile=False*)
next unused multi-file address, or None if newfile cannot create

open(*n*)
open n-th file of family

seek(*addr*)
seek to multi-file address, opening alternate file if needed

split_address(*addr*)
return file index, address for a multifile address

tell()
return current multi-file address

zero_address(*n=None*)
multifile address of first byte in current or n-th file

qnd.generic.**opener**(*filename*, *mode*, ***kwargs*)
Generic file or file family opener.

Parameters

- **filename** (*str*) – Name of file to open. See notes below for family conventions.
- **mode** (*str*) – One of 'r' (default, read-only), 'r+' (read-write, must exist), 'a' (read-write, create if does not exist), 'w' (create, clobber if exists), 'w-' (create, fail if exists).
- ****kwargs** – Other keywords. This opener consumes one item from kwargs:
- **nextaddr_mode** (*int*) – Affects setting of nextaddr for families opened with 'a' or 'r+' mode. 0 (default) sets nextaddr to the end of the final existing file, 1 sets nextaddr to 0 (beginning of first file), and 2 sets nextaddr to the beginning of the next file after all existing files.

Returns

- *handle* –

  A file handle implementing the generic interface, consisting of:

```
handle.callbacks(flusher, initializer)
addr = handle.next_address()  # next unused address
f = handle.seek(addr)  # return ordinary file handle at addr
f = handle.open(n)  # open nth file, calling initializer(f)
handle.flush()  # make file readable, calling flusher(f)
# flush() restores next_address to value on entry
handle.close()  # flush if necessary, then close
```

- **nexisting** (*int*) – Number of existing paths matching *filename*.

Notes

The *filename* may be an iterable, one string per file in order. The sequence may extend beyond the files which actually exist for 'r+', 'a', 'w', or 'w-' modes.

Alternatively *filename* specifies a family if it contains shell globbing wildcard characters. Existing matching files are sorted first by length, then alphabetically (ensuring that 'file100' comes after 'file99', for example). If there is only a single wildcard group, it also serves to define a sequence of future family names beyond those currently existing for 'r+', 'a', 'w', or 'w-' modes. A '?' pattern is treated the same as a '[0-9]' pattern if all its matches are digits or if the pattern matches no existing files. Similarly, a '*' acts like the minimum number of all-digit matches, or three digits if there are no matches.

A single filename may also contain a `%d` or `%0nd` print format directive, which will be converted to the corresponding number of `[0-9]` glob patterns.

genindex	modindex	search

1.5 qnd.h5f module

QnD wrapper for h5py HDF5 interface.

`qnd.h5f.`**`openh5`**(*filename*, *mode='r'*, *auto=1*, ***kwargs*)

Open HDF5 file using h5py, but wrapped as a QnD QGroup.

Parameters

- **`filename`** (*`str`*) – Name of file to open. If filename contains %d format directive, open a family of files (the memb_size keyword controls file size).
- **`mode`** (*`str`*) – One of 'r' (default, read-only), 'r+' (read-write, must exist), 'a' (read-write, create if does not exist), 'w' (create, clobber if exists), 'w-' (create, fail if exists).
- **`auto`** (*`int`*) – The intial state of auto-read mode. If the QGroup handle returned by openh5 is *f*, then `f.varname` reads an array variable, but not a subgroup when auto=1, the default. With auto=0, the variable reference reads neither (permitting later partial reads in the case of array variables). With auto=2, a variable reference recursively reads subgroups, bringing a whole tree into memory.
- **`**kwargs`** – Other keywords passed to h5py.File constructor. Note that the driver='family' keyword is implicit if filename contains %d.

Returns **f** – A file handle implementing the QnD interface.

Return type *QGroup*

genindex	modindex	search

1.6 qnd.lazy module

Lazy file type identification and permission filters.

`qnd.lazy.`**`openb`**(*filename*, *mode='r'*, *auto=1*, ***kwargs*)
Open PDB or HDF5 or netCDF or some othe binary file known to QnD.

This imports the required package as necessary. See `qnd.pdbf.openpdb` or `qnd.h5f.openh5` or some other actual open function for argument details.

`qnd.lazy.`**`permfilter`**(**names*)
Given a glob string or list of filenames, exclude unreadable ones.

Parameters **`names`** (*`str or list of str`*) – An individual filename or a list of filenames, which may include globbing characters like '*' or '?'. Accepts any number of such arguments.

Returns

- **names** (*list of str*) – The subset of all the specified names for which you have read permission (and which exist). The list may be empty.
- *Globbed results are sorted first by length, then alphabetically,*
- *so that, for example, name100 follows name99. Note, however, that*
- *name000 also follows name99.*

genindex	modindex	search

1.7 qnd.ncf module

QnD netCDF3 interface.

`qnd.ncf.`**`opennc`**(*filename*, *mode='r'*, *auto=1*, ***kwargs*)
Open netCDF-3 file returning a QnD QGroup.

A netCDF-3 file differs from other self-describing binary file formats because no data addresses can be known until every variable to be stored is declared. Therefore, when writing a netCDF-3 file, you must declare every variable before you can begin writing anything.

The qnd API is somewhat at odds with this semantics because it encourages you to declare and write each variable in a single step. The native netCDF-3 API forces you to declare everything, then call an *enddef* method to complete all variable declarations and permit you to begin writing data. The qnd.ncf backend uses the first call to the ordinary qnd *flush* method to emulate the netCDF-3 *enddef* mode switch – thus nothing will be written to the file until the first call to *flush*. To minimize the difference between ncf and other qnd backends, if you do use the usual qnd declare-and-write idiom, the ncf backend will save the variable value in memory until the first *flush* call, which will trigger the actual writing of all such saved values.

Note that closing the file flushes it, so that is also a viable way to finish a netCDF-3 file. Furthermore, when you overwrite any record variable in *recording* mode, ncf will implicitly *flush* the file, since no new variables can be declared after that.

Note that you use the standard QnD API, a copy of every variable you write to the file until you begin the second record will be kept in memory, which could potentially be a problem. If you wish to declare all variables before writing anything, so that your code is aligned with the netCDF API, do something like this:

```
f = opennc("myfile??.nc", "w")  # wildcards expand to 00, 01, 02, ...
# declare non-record variables from in-memory arrays
f.nrvar1 = nrvar1.dtype, nrvar1.shape
f.nrvar2 = nrvar2.dtype, nrvar2.shape
# declare record variables from in-memory arrays
f.recording(1)
f.rvar1 = rvar1.dtype, rvar1.shape
f.rvar2 = rvar2.dtype, rvar2.shape
# flushing the file is equivalent to netCDF ENDDEF mode switch
f.flush()
# now write the current values of all the variables
f.nrvar1 = nrvar1
f.nrvar2 = nrvar2
# writing the record variables writes their values for first record
f.rvar1 = rvar1
f.rvar2 = rvar2
# change values of record variables and write the second record
f.rvar1 = rvar1
f.rvar2 = rvar2
# when you've written all records, close the file
f.close()
```

Parameters

- **filename** (*str*) – Name of file to open. See notes below for file family.
- **mode** (*str*) – One of 'r' (default, read-only), 'r+' (read-write, must exist), 'a' (read-write, create if does not exist), 'w' (create, clobber if exists), 'w-' (create, fail if exists).
- **auto** (*int*) – The intial state of auto-read mode. If the QGroup handle returned by openh5 is *f*, then `f.varname` reads an array variable, but not a subgroup when auto=1, the default. With auto=0, the variable reference reads neither (permitting later partial reads in the case of array variables). With auto=2, a variable reference recursively reads subgroups, bringing a whole tree into memory.
- ****kwargs** – Other keywords. The maxsize keyword sets the size of files in a family generated in recording==1 mode; a new file will begin when the first item in a new record would begin beyond *maxsize*. The default maxsize is 128 MiB (134 MB). The v64 keyword, if provided and true, causes new files to be created using the 64-bit netCDF format; the default is to create 32-bit files. (But a file family always uses a single format.) The nextaddr_mode keyword can be used to indicate whether the next new record in 'a' or 'r+' mode should go into a new file. The default behavior is that it should, which is the pdbf module default; this is nextaddr_mode true. Use nextaddr_mode=0 to continue filling the final existing file until maxsize.

Returns **f** – A file handle implementing the QnD interface.

Return type *QGroup*

Notes

The *filename* may be an iterable, one string per file in order. The sequence may extend beyond the files which actually exist for 'r+', 'a', 'w', or 'w-' modes.

Alternatively *filename* specifies a family if it contains shell globbing wildcard characters. Existing matching files are sorted first by length, then alphabetically (ensuring that 'file100' comes after 'file99', for example). If there is only a single wildcard group, it also serves to define a sequence of future family names beyond those currently existing for 'r+', 'a', 'w', or 'w-' modes. A '?' pattern is treated the same as a '[0-9]' pattern if all its matches are digits or if the pattern matches no existing files. Similarly, a '*' acts like the minimum number of all-digit matches, or three digits if there are no matches.

genindex	modindex	search

1.8 qnd.pdbdump module

Low level functions to create and write a PDB file.

genindex	modindex	search

1.9 qnd.pdbf module

Pure python QnD wrapper for PDB files.

Information on the PDB file format is somewhat hard to come by. Try the SILO github repo https://github.com/LLNL/Silo esp. src/pdb/ and src/score/ dirs. Also the end of the QND file pdbparse.py has a long comment with a detailed description.

Note that yorick-generated PDB files are version II, not version III. Also, yorick pointers are written (by default) in a yorick-specific format. Since yorick readability has held back many application codes (the LEOS library and LLNL rad-hydro codes used for ICF design), most of the dwindling legacy PDB files are version II. Hence, this implementation focuses on the version III format, and the version I format is supported only for reading.

Furthermore, this implementation only supports IEEE 754 4 and 8 byte floating point formats, since those are the only unambiguous floating point formats supported by numpy. Fortunately, this covers all modern PDB files likely to show up in practice, so we have no significant incentive to do the work required to support exotic formats.

`qnd.pdbf.`**`openpdb`**(*filename*, *mode='r'*, *auto=1*, ***kwargs*)

Open PDB file or family, and wrap it in a QnD QGroup.

Parameters

- **`filename`** (*`str`*) – Name of file to open. See notes below for file family.
- **`mode`** (*`str`*) – One of 'r' (default, read-only), 'r+' (read-write, must exist), 'a' (read-write, create if does not exist), 'w' (create, clobber if exists), 'w-' (create, fail if exists).
- **`auto`** (*`int`*) – The intial state of auto-read mode. If the QGroup handle returned by openh5 is *f*, then `f.varname` reads an array variable, but not a subgroup when auto=1, the default. With auto=0, the variable reference reads neither (permitting later partial reads in the case of array variables). With auto=2, a variable reference recursively reads subgroups, bringing a whole tree into memory.
- **`**kwargs`** – Other keywords. The *maxsize* keyword sets the size of files in a family generated in `recording==1` mode; a new file will begin when the first item in a new record would begin beyond *maxsize*. The default maxsize is 128 MiB (134 MB). The *order* keyword can be '>' or '<' to force the byte order in a new file; by default the byte order is the native order. File families always have the same order for every file, so *order* is ignored if any files exist.

Returns **f** – A file handle implementing the QnD interface.

Return type *QGroup*

Notes

The *filename* may be an iterable, one string per file in order. The sequence may extend beyond the files which actually exist for 'r+', 'a', 'w', or 'w-' modes.

Alternatively *filename* specifies a family if it contains shell globbing wildcard characters. Existing matching files are sorted first by length, then alphabetically (ensuring that 'file100' comes after 'file99', for example). If there is only a single wildcard group, it also serves to define a sequence of future family names beyond those currently existing for 'r+', 'a', 'w', or 'w-' modes. A '?' pattern is treated the same as a '[0-9]' pattern if all its matches are digits or if the pattern matches no existing files. Similarly, a '*' acts like the minimum number of all-digit matches, or three digits if there are no matches.

| genindex | modindex | search |

1.10 qnd.pdbparse module

Pure python PDB file format parsing.

Parse PDB metadata. The PDB file format was designed by Stewart Brown at Lawrence Livermore National Laboratory in the 1990s.

PDB is widely used at LLNL for restart and post-processing data produced by large radiation-hydrodynamics simulation codes. The PDB format can describe and store named arrays of any data type representable in the C programming language that is derived from one of the primitive C data types char, short, int, long, float, or double or pointer to a representable type. The format was extended to handle primitive integer or floating point numbers of any size, such as long long or long double or 16 bit floating point data. The format was also extended to organize the named arrays into groups like HDF5. The metadata is text with a few embedded ASCII control characters, written at the end of the file, and intended to be parsed and held in memory when the file is opened.

class qnd.pdbparse.**PDBChart**(*root*)

Bases: object

PDB structure chart contains all information about data types.

add_primitive(*name*, *desc*)

desc is (size, order, align) or (size, order, align, fpbits)

add_struct(*name*, *members*)

members is OrderedDict, name –> typename, shape

find_or_create(*dtype*)

Find or create stype –> (stype, align, typename).

read_special(*f*, *typename*, *value*)

Recursively read pointer values.

use(*dtype*)

Find or create stype –> (stype, align, typename, nopartial).

qnd.pdbparse.**parser**(*handle*, *root*, *index=0*)

Parse PDB file with the given MultiFile handle and PDBGroup group.

| genindex | modindex | search |

1.11 qnd.utils module

1.12 Example Write HDF5 File

This script is an example of writing single values and records to a file in HDF5 format. It is located at `[QND repo]/qnd/examples/example_write_h5.py`.

```
'''
example_write_h5.py demos how to write single values and multiple records
into an hdf5 file.
It writes to the current directory, so run it from one where you have permission to write.
'''
import numpy as np
from qnd.h5f import openh5

fname = "foo.h5"

with openh5(fname, "w") as f:

    # write a single value:
    f.x = 3

    # turn on recording and write two records:
    f.recording(1)
    f.tm = 0
    f.y = 0.1
    f.z = np.arange(3)

    f.tm = 1
    f.y = 0.2
    f.z = np.arange(3)+1

# reopen file to append data:
with openh5(fname, "a") as f:

    # append a single new variable to the file
    f.a = 10

    # write one new record:
    f.recording(1)
    f.tm = 2
    f.y = 0.3
    f.z = np.arange(3)+2

```

1.13 Example Write PDB Files

This script is an example of writing single values and records to a file in PDB format. It is located at `[QND repo]/qnd/examples/example_write_pdb.py`.

```
'''
example_write_pdb.py demos how to write single values and multiple records
```

(continues on next page)

(continued from previous page)

```
into a family of pdb files.
It writes to the current directory, so run it from one where you have permission to write.
'''
import numpy as np
from qnd.pdbf import openpdb

with openpdb("foo000.pdb", "w") as f:

    # write a single value:
    f.x = 3

    # turn on recording and write two records:
    f.recording(1)
    f.tm = 0
    f.y = 0.1
    f.z = np.arange(3)

    f.tm = 1
    f.y = 0.2
    f.z = np.arange(3)+1

# append a single new variable to the file:
with openpdb("foo000.pdb", "a") as f:
    f.a = 10

# open a new file in the family and write one new record:
with openpdb("foo001.pdb", "w") as f:
    f.recording(1)
    f.tm = 2
    f.y = 0.3
    f.z = np.arange(3)+2

```

1.14 Example Read Files

This script reads the contents of both `example_write*.py` scripts (which should both be run prior to running this script) and prints the file contents:

```
> python example_read.py

Opening file(s): foo*.pdb
x: 3
tm: [0, 1, 2]
y: [0.1, 0.2, 0.3]
z: [array([0, 1, 2]), array([1, 2, 3]), array([2, 3, 4])]
a: 10

Opening file(s): foo.h5
a: 10
tm: [0, 1, 2]
x: 3
y: [0.1, 0.2, 0.3]
z: [array([0, 1, 2]), array([1, 2, 3]), array([2, 3, 4])]
```

It is located at `[QND repo]/qnd/examples/example_read.py`.

```
'''
example_read.py opens and lists the contents of files produced by
example_write_pdb.py and example_write_h5.py
'''
import numpy as np
from qnd.lazy import openb
from qnd.frontend import QList

fnames = ["foo*.pdb", "foo.h5"]

for fname in fnames:
    print("\nOpening file(s): {}".format(fname))

    with openb(fname, "r") as f:
        for k in list(f):
            val = f[k]
            if type(f[k]) is QList:
                val = val[:]
            print(f"{k}: {val}")
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

q

Index

R

S

T

U

Z