Querylist Documentation

Release 0.2.0

Thomas Welfley

January 24, 2015

Contents

Sick of for loop + conditional soup when dealing with complicated lists? Querylist is here to help.

This package provides a data structure called a QueryList, an extension of Python's built in list data type that adds django ORM-eseque filtering, exclusion, and get methods. QueryLists allow developers to easily query and retrieve data from complex lists without the need for unnecessarily verbose iteration and selection cruft.

The package also provides BetterDict, a backwards-compatible wrapper for dictionaries that enables dot lookups and assignment for key values.

CHAPTER 1

Contents

1.1 Installation

Querylist can be installed like any other python package:

```
$ pip install querylist
```

Once installed, both QueryList and BetterDict can be imported directly from the querylist module.

```
>>> from querylist import BetterDict, QueryList
```

QueryList is currently only tested against Python 2.6 and 2.7, though it should work in 2.5 as well.

1.2 BetterDict

A BetterDict is a data structure that is backwards compatible with Python's built in dict type. It has all of the usual dict methods and can still preform index based lookup and assignment of key values. Even an equal to comparison of a BetterDict instance invoked with the same data as a dict instance will evaluate to true.

In addition to standard dict functionality, BetterDicts also allows dot lookup and assignment of key values.

```
>>> from querylist import BetterDict
>>> a = BetterDict()
>>> b = dict()
>>> a.cats = 18
>>> a['dogs'] = 372
>>> a.hedgehogs = 19
>>> b['cats'] = 18
>>> b['dogs'] = 372
>>> b['hedgehogs'] = 19
>>> a == b
True
>>> dir(a)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
 '__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
```

```
'__subclasshook__', 'cats', 'clear', 'copy', 'dogs', 'fromkeys', 'get',
'has_key', 'hedgehogs', 'items', 'iteritems', 'iterkeys', 'itervalues',
'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

1.2.1 Dot lookups and assignments

Dot lookups that correspond to keys will return the key's value

```
>>> foo = BetterDict()
>>> foo['yay'] = 1
>>> foo.yay
1
```

An assignment will update that key's value:

```
>>> foo.yay = 0
>>> foo['yay']
0
```

Assignment of an attribute that has not yet been added as a key to the BetterDict will add the key/value pair.

```
>>> foo.yeah = True
>>> foo['yeah']
True
```

However, a lookup for an attribute that is neither a key nor a normal dict attribute will raise an `AttributeError`:

```
>>> foo.bar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "querylist/betterdict.py", line 6, in __getattr__
    raise AttributeError
AttributeError
```

1.2.2 Nested Dictionaries

A nested dictionary's keys are also accessible via dot lookup. BetterDict achieves this by converting member dicts to BetterDicts:

Consider the following BetterDict definition:

```
>>> c = BetterDict({
...     'foo': 1,
...     'bar': {
...         'a': 0,
...         'b': 1,
...         'c': 2,
...         'another': {
...             'd': 0,
...             'e': 1,
...             'f': 2,
...         }
...     },
... })
```

`c.bar` will return the BetterDict at `c['bar']`:

```
>>> c.bar
{'a': 0, 'c': 2, 'b': 1, 'another': {'e': 1, 'd': 0, 'f': 2}}
```

Similarly, `c.bar.another.d` will descend down to `c['bar']['another']['d']` and an assignment using the same identifier chain will update that value:

```
>>> c.bar.another.d
0
>>> c.bar.another.d = True
>>> c.bar.another
{'e': 1, 'd': True, 'f': 2}
```

1.2.3 Resolving key name / dict attribute conflicts (the _bd_ attribute)

When a BetterDict has a key that conflicts with the name of a standard dict attribute, the BetterDict does not overwrite the standard attribute. Doing so would break backwards compatibility for BetterDicts with specific key/value pairs.

```
>>> problem = BetterDict({'update': '3 days ago'})
>>> problem.update
<built-in method update of BetterDict object at 0x7fb9f047b610>
>>> problem['update']
'3 days ago'
```

As a work around for this issue, all BetterDicts have an attribute named _bd_, which is protected from being overwritten just like standard dict attributes. The _bd_ attribute allows dot lookup and assignment of all a BetterDict's keys regardless of whether or not their name conflicts with a dict attribute.

```
>>> problem._bd_.update
'3 days ago'
>>> problem._bd_.update = '4 days ago'
>>> problem['update']
'4 days ago'
```

1.3 QueryList

class querylist.**QueryList**(*data=None*, *wrapper=<class 'querylist.betterdict.BetterDict'>*, *wrap=True*)

A QueryList is an extension of Python's built in list data structure that adds easy filtering, excluding, and retrieval of member objects.

```
>>> from querylist import QueryList
>>> sites = QueryList(get_sites())
>>> sites.exclude(published=True)
[{'url': 'http://site3.tld/', 'published': False}]
```

Keywrod arguments:

- data – an iterable reprsenting the data that to be to queried.
- wrapper – a callable that can convert data's elements to objects that are compatbile with QueryList
- wrap – Boolean toggle to indicate whether or not to call wrapper on each element in data on instantiation. Set to false if data's elements are already compatible with QueryList.

1.3.1 Instantiation

By default, QueryLists expect to be instantiated with lists of dictionaries or dictionary-like data because it attempts to convert the elements to BetterDicts:

```
>>> a = QueryList([{'foo': 1}, {'foo': 2}])
>>> b = QueryList([(('foo', 1),),(('foo', 2),) ])
>>> a == b
True
>>> a.get(foo__lt=2)
{'foo': 1}
```

Fortunately, QueryLists can also be instantiated with iterables returning any objects that support dot lookups. To invoke a QueryList with a custom wrapper called `myclass`, one could write the following:

```
>>> c = QueryList(mydata, wrapper=myclass, wrap=False)
```

`wrap=False` in the above examples tells the QueryList that it doesn't need to convert the elements of mydata to myclass. If wrap had been true, something similar to `[myclass(x) for x in mydata]` would have been executed.

If `mydata` had been a list of data that can be converted to a custom object, then `wrap=True` (the default behavior) would have been appropriate.

```
>>> d = QueryList(mydata, wrapper=myclass)
```

`wrapper` can be any callable that returns a QueryList compatible object. It doesn't need to be a class.

It is also possible to instantiate an empty QueryList and add objects to it as needed:

```
>>> a = QueryList()
```

1.3.2 Querying

All objects

You can iterate over a QuerySet like a normal list and it will return the data that was used to instantiate it (the data will be wrapped, if `wrapped=True` when the list was instantiated).

```
>>> a = QueryList(my_data)
>>> [item for item in a]
```

Filtering and Excluding

QueryLists provide two methods for filtering and excluding objects from a QueryList: `filter()` and `exclude()`. `filter()` will return a QueryList containing all objects in the list that match the passed conditions, and `exclude()` will return a QueryList containing the subset of the original QueryList that doesn't match the passed conditions.

Both methods accept keyword argument/value pairs, where the keyword is a field lookup and the value is the value to compare that field to. For example, `id=4` would match all objects with an id property equal to 4. See *Field lookups* for more information.

`QueryList.`**`filter`**`(`*`**kwargs`*`)`

Generates a QueryList containing the subset of objects from this QueryList that match the provided set of field lookups.

The following example returns the subset of a QueryList named `site_list` where published is equal to False:

```
>>> site_list.filter(published=True)
[{'url': 'http://site1.tld/',...}, {...}],
```

Similarly, in the next example, `filter()` returns the subset of objects where object.meta.keywords contains the string 'kittens' and where the id property is greater than 100.

```
>>> site_list.filter(meta__keywords__contains='kittens', id__gt=100)
[{'url': 'http://site101.tld/',...}, {...}],
```

If no objects match the provided field lookups, an empty QueryList is returned.

```
>>> site_list.filter(id__gte=1000, published=False)
[]
```

QueryList.**exclude**(***kwargs*)

Generates a QueryList containing the subset of objects from this QueryList that do **not** match the provided field lookups.

The following example returns the subset of a QueryList named `site_list` where the id is greather than 1000.

```
>>> site_list.exclude(id__gt=1000)
[{'url': 'http://site1001.tld/',...}, {...}],
```

In the next example, `exclude()` returns the subset of objects from site_list that aren't published and don't have "test" in their title

```
>>> site_list.exclude(published=True, title__icontains="test")
[{'url': 'http://site1.tld/',...}, {...}]
```

If all objects match the provided field lookups, then an empty QueryList is returned:

```
>>> site_list.exclude(id__gt=0)
[]
```

Chaining

QueryList methods that return QueryLists (*filter()* and *exclude()*) can be chained together to form more complex queries:

```
>>> QueryList(sites).filter(published=False).exclude(meta__keywords__contains="kittens")
[]
```

Retrieving a single object

In addition to providing methods for filtering or excluding objects, QueryLists provide a method for retrieving specific objects:

QueryList.**get**(***kwargs*)

Returns the first object encountered that matches the specified lookup parameters.

```
>>> site_list.get(id=1)
{'url': 'http://site1.tld/', 'published': False, 'id': 1}
>>> site_list.get(published=True, id__lt=3)
{'url': 'http://site1.tld/', 'published': True, 'id': 2}
>>> site_list.filter(published=True).get(id__lt=3)
{'url': 'http://site1.tld/', 'published': True, 'id': 2}
```

If the QueryList contains multiple elements that match the criteria, only the first match will be returned. Use `filter()` to retrieve the entire set.

If no match is found in the QueryList, the method will raise a `NotFound` exception.

```
>>> site_list.get(id=None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "querylist/querylist.py", line 55, in get
querylist.querylist.NotFound: Element with specified attributes not
found.
```

1.3.3 Field lookups

A field lookup consists of a name corresponding to an object attribute and a value to compare that attribute against. The following is a field lookup for all QueryList object whose id property is equal to 1000: `id=1000`.

Similarly, `title="Go Team!"` would match all objects with the title team.

Field lookups can be combined arbitrarily: `id=1000, title="Go team!"` would match all objects with the id 1000 and the title "Go Team!".

Properties of properties

Field lookups can extend to properties of properties (and so on). Simply replace the dot operator that one would normally use with a double underscore. `meta__description="Cats"`, for example, would match against all objects in the QueryList where object.meta.description==”Cats.”

Similarly, `meta__keywords__count=4` would match against all objects in the QueryList where object.meta.keywords.count==4.

Comparators

Field lookups can end with an optional comparator designation that indicates that the lookup should do something other than an exact comparison: `attribute__<comparator>`.

A field lookup that does a case insensitive match against the title attribute would look like: `title__iexact="cats"`.

Querylist ships with a number of comparators:

exact

Returns True if the attribute and the specified value are equal.

iexact

Converts both the attribute and the specified value to lowercase and returns True if the values are equal.

Both the attribute and the specified value must be strings.

contains

Returns True if the specified value is `in` the attribute value. This works with strings and lists.

icontains

Converts both the attribute and the specified value to lowercase and returns True if the specified value is `in` the attribute value.

in

Returns True if the attribute is `in` the specified iterable

This requires the specified value to be some iterable.

startswith

Returns true if the attribute value starts with the specified value.

This requires the attribute value and specified value to be strings.

istartswith

Case insensitive startswith.

endswith

Returns true if the attribute value ends with the specified value.

This requires the attribute value and specified value to be strings.

iendswith

Case insensitive endswith.

regex

Returns True if the attribute value matches the specified regular expression.

iregex

Case insensitive regex.

gt

Returns True if the attribute value is greater than the specified value.

gte

Returns True if the attribute value is greater than or equal to the specified value.

lt

Returns True if the attribute value is less than the specified valued.

lte

Returns True if the attribute value is less than or equal to the specified value.

call

Invokes the specified value as a callable, passing it the attribute value. The result is returned. If the result is falsey, this test will fail. If the result is truthy it will pass.

1.3.4 Aggregation

`QueryList.`**`count`**
: Returns the nubmer of objects in the QueryList.

1.3.5 Backwards compatibility

QueryLists are intended to be a drop in replacement for lists of dictionaries. Because QueryLists and their default wrapper (BetterDicts) are backwards compatible with lists and dicts respectively, a developer can drop them into existing projects without changing the existing behavior.

Consider a user class that returns a list of sites:

```
class User(object):
    def get_sites():
        """Returns a list of the user's sites."""
        return Site(self.id).get_all_sites()
```

If dictionaries are being used to represent sites, we can change the definition of `get_sites()` as follows without impacting any existing functionality:

```
def get_sites():
    """Returns a list of the user's sites."""
    return QueryList(Site(self.id).get_all_sites())
```

The new `get_sites()` will be backwards compatible with its old definition, but for any new code written, the developer can use QueryList and BetterDict functionality to their heart's content.

1.4 Changelog

0.2.0

- Added a new field lookup called *call*. It uses the passed value as a callable to test the specified attribute of QueryList objects.
- Fixed a bug in the QueryList implementation that would cause querylist+querylist addition and querylist+list addition to return lists. These operations now return a new QueryList instance containing the combined data.

0.1.0

- Renamed QueryList's 'limit' method to 'filter' so that the QueryList API is more consistent with Django's QuerySets.

0.0.1

- Initial relase

CHAPTER 2

Indices and tables

- *genindex*
- *search*

Index