asq Documentation

Release 1.3a

Sixty North

Dec 04, 2017

Contents

asq is a Python package for specifying and performing efficient queries over collections of Python objects using a fluent interface. It is licensed under the MIT License

CHAPTER 1

Contents

1.1 Front Matter

1.1.1 Copyright

`asq`

1.1.2 Official Website

https://github.com/rob-smallshire/asq

1.1.3 License

1.2 Narrative Documentation

Read this to learn how to use asq.

1.2.1 asq Introduction

asq implements a chained declarative style queries for Python *iterables*. This provides an alternative to traditional for loops or comprehensions which are ubiquitious in Python. Query methods can offer the following advantages over loops or comprehensions:

1. **Concision**: asq query expressions can be to the point, especially when combining multiple queries.
2. **Readability**: Chained asq query operators can have superior readability to nested or chained comprehensions. For example, multi-key sorting is much clearer with asq than with other approaches.
3. **Abstraction**: Query expressions in asq decouple query specification from the execution mechanism giving more flexibility with how query results are determined, so for example queries can be executed in parallel with minimal changes.

More complex queries tend to show greater benefits when using asq. Simple transformations are probably best left as regular Python comprehensions. It's easy to mix and match asq with comprehensions and indeed any other Python function which produces or consumes *iterables*.

1.2.2 Installing asq

asq is available on the Python Package Index (PyPI) and can be installed with pip:

```
$ pip install asq
```

Alternatively, you can download and unpack the source distribution from the asq **'downloads page'_** or PyPI. You should then unpack the source distribution into a temporary directory and run the setup script which will install asq into the current Python environment, for example:

```
$ tar xzf asq-1.0.tar.gz
$ cd asq-1.0
$ python setup.py install
```

If you are using Python 2.6 you will also need to install the back-ported ordereddict module which was introduced in Python 2.7.

1.2.3 Diving in

A few simple examples will help to illustrate use of asq. We'll need some data to work with, so let's set up a simple list of student records, where each student is represented by a dictionary:

```
students = [dict(firstname='Joe', lastname='Blogs', scores=[56, 23, 21, 89]),
            dict(firstname='John', lastname='Doe', scores=[34, 12, 92, 93]),
            dict(firstname='Jane', lastname='Doe', scores=[33, 94, 91, 13]),
            dict(firstname='Ola', lastname='Nordmann', scores=[98, 23, 98, 87]),
            dict(firstname='Kari', lastname='Nordmann', scores=[86, 37, 88, 87]),
            dict(firstname='Mario', lastname='Rossi', scores=[37, 95, 45, 18])]
```

To avoid having to type in this data structure, you can navigate to the root of the unpacked source distribution of asq and then import it from pupils.py in the examples directory with:

```
$ cd asq/examples/
$ python
Python 2.6.2 (r262:71605, Apr 14 2009, 22:40:02) [MSC v.1500 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from pupils import students
```

Now we can import the query tools we need. We'll start with the most commonly used import from `asq` which is the `query` initiator:

```
>>> from asq import query
```

The `query` initiator allows us to perform queries over any Python iterable, such as the `students` object we imported.

Let's start by creating a simple query to find those students who's first names begin with a letter 'J':

```
>>> query(students).where(lambda student: student['firstname'].startswith('J'))
Queryable(<filter object at 0x00000000031D9B70>)
```

To dissect this line and its result left to right, we have:

1. A call to the `query(students)`. Here `query()` is a query *initiator* - a factory function for creating a Queryable object from, in this case, an iterable. The `query()` function is the key entry point into the query system (although there are others).
2. A method call to `where()`. Where is one of the `asq` query operators and is in fact a method on the Queryable returned by the preceding call to `query()`. The `where()` query operator accepts a single argument, which is a callable predicate (*i.e.* returning either True or False) function which which each element will be tested.
3. The predicate passed to `where()` is defined by the expression `lambda student: student['firstname'].startswith('J')` which accepts a single argument `student` which is the element being tested. From the `student` dictionary the first name is extracted and the built-in string method `startswith()` is called on the name.
4. The result of the call is a Queryable object. Note that no results have yet been produced - because the query has not yet been executed. The Queryable object contains all the information required to execute the query when results are required.

Initiators

All query expressions begin with query *initiator*. Initiators are the entry points to `asq` and are to be found the in the `initiators` submodule. The most commonly used `query` initiator is also availble from the top-level `asq` namespace for convenience. All initiators return Queryables on which any query method can be called. We have already seen the `query()` initiator in use. The full list of available query initiators is:

Initiator	Purpose
`query(iterable)`	Make a Queryable from any iterable
`integers(start, count)`	Make a Queryable sequence of consecutive integers
`repeat(value, count)`	Make a Queryable from a repeating value
`empty()`	Make a Queryable from an empty sequence

When is the query evaluated?

In order to make the query execute we need to iterate over the Queryable or chain additional calls to convert the result to, for example, a list. We'll do this by creating the query again, but this time assigning it to a name:

```
>>> q = query(students).where(lambda student: student['firstname'].startswith('J'))
>>> q
Queryable(<filter object at 0x00000000031D9BE0>)
>>> q.to_list()
[{'lastname': 'Blogs', 'firstname': 'Joe', 'scores': [56, 23, 21, 89]},
 {'lastname': 'Doe', 'firstname': 'John', 'scores': [34, 12, 92, 93]},
 {'lastname': 'Doe', 'firstname': 'Jane', 'scores': [33, 94, 91, 13]}]
```

Most of the `asq` query operators like `where()` use so-called deferred execution whereas others which return non-Queryable results use immediate execution and force evaluation of any pending deferred operations.

Queries are executed when the results are realised by converting them to a concrete type such as a list, dictionary or set, or by any of the query operators which return a single value.

Query chaining

Most of the query operators can be composed in chains to create more complex queries. For example, we could extract and compose the full names of the three students resulting from the previous query with:

```
>>> query(students).where(lambda s: s['firstname'].startswith('J'))        \
...                .select(lambda s: s['firstname'] + ' ' + s['lastname']) \
...                .to_list()
['Joe Blogs', 'John Doe', 'Jane Doe']
```

Note: The backslashes above are Python's line-continuation character, used here for readability. They are not part of the syntax of the expression.

If we would like our results sorted by the students' minimum scores we can use the Python built-in function `min()` with the `order_by` query operator:

```
>>> query(students).where(lambda s: s['firstname'].startswith('J'))        \
...                .order_by(lambda s: min(s['scores']))                   \
...                .select(lambda s: s['firstname'] + ' ' + s['lastname']) \
...                .to_list()
['John Doe', 'Jane Doe', 'Joe Blogs']
```

Query nesting

There is nothing to stop us initiating a sub-query in the course of defining a primary query. For example, to order the students by their average score we can invoke the `query()` initiator a second time and chain the `average()` query operator to determine the mean score to pass to `order_by()`:

```
>>>  query(students).order_by(lambda s: query(s['scores']).average())       \
...                .where(lambda student: student['firstname'].startswith('J')) \
...                .select(lambda s: s['firstname'] + ' ' + s['lastname'])      \
...                .to_list()
['Joe Blogs', 'John Doe', 'Jane Doe']
```

Selectors

Many of the query operators, such as `select()`, `order_by` or `where()` accept selector callables for one or more of their arguments. Typically such selectors are used to *select* or *extract* a value from an element of the query sequence.

Selectors can be any Python callable and examples of commonly used selectors are demonstrated below. In addition, `asq` provides some selector factories as a convenience for generating commonly used forms of selectors.

Most of the selectors used in `asq` are unary functions, that is, they take a single positional argument which is the value of the current element. However, some of the query operators do require selectors which take two arguments; these cases are noted in the API documentation.

Lambdas

Lambda is probably the most frequently used mechanism for specifying selectors. This example squares each element:

```
>>> numbers = [1, 67, 34, 23, 56, 34, 45]
>>> query(numbers).select(lambda x: x**2).to_list()
[1, 4489, 1156, 529, 3136, 1156, 2025]
```

Functions

Sometime the selector you want cannot be easily expressed as a lambda, or it is already available as a function in existing code, such as the standard library.

In this example we use the built-in `len()` function as the selector:

```
>>> words = 'The quick brown fox jumped over the lazy dog'.split()
>>> words
['The', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
>>> query(words).select(len).to_list()
[3, 5, 5, 3, 6, 4, 3, 4, 3]
```

Unbound methods

Unbound methods are obtained by referencing the method of a *class* rather than the method of an *instance*. That is, the *self* parameter passed as the first argument of a method has not yet been specified. We can pass any unbound method which takes only a single argument *including* the normally implicit *self* as a selector.

In this example, we use an unbound method `upper()` of the built-in string class:

```
>>> words = ["the", "quick", "brown", "fox"]
>>> query(words).select(str.upper).to_list()
['THE', 'QUICK', 'BROWN', 'FOX']
```

This has the effect of making the method call *on* each element in the sequence.

Bound methods

Bound methods are obtained by referencing the method of an *instance* rather than the method of a class. That is, the instance referred to by the *self* parameter passed as the first argument of a method has already been determined.

To illustrate, here we create a Multiplier class instances of which multiply by a factor specified at initialization when the `multiply` method is called:

```
>>> numbers = [1, 67, 34, 23, 56, 34, 45]
>>>
>>> class Multiplier(object):
...     def __init__(self, factor):
...         self.factor = factor
...     def multiply(self, value):
...         return self.factor * value
...
>>> five_multiplier = Multiplier(5)
>>> times_by_five  = five_multiplier.multiply
>>> times_by_five
<bound method Multiplier.multiply of <__main__.Multiplier object at␣
↪0x0000000002F251D0>>
>>>
>>> query(numbers).select(times_by_five).to_list()
[5, 335, 170, 115, 280, 170, 225]
```

This has the effect of passing each element of the sequence in turn as an argument to the bound method.

Selector factories

Some selector patterns crop up very frequently and so `asq` provides some simple and concise selector factories for these cases. Selector factories are themselves functions which return the actual selector function which can be passed in turn to the query operator.

Selector factory	Created selector function
`k_(key)`	`lambda x: x[key]`
`a_(name)`	`lambda x: getattr(x, name)`
`m_(name, *args, **kwargs)`	`lambda x: getattr(x, name)(*args, **kwargs)`

Key selector factory

For our example, we'll create a list of employees, with each employee being represented as a Python dictionary:

```
>>> employees = [dict(firstname='Joe', lastname='Bloggs', grade=3),
...              dict(firstname='Ola', lastname='Nordmann', grade=3),
...              dict(firstname='Kari', lastname='Nordmann', grade=2),
...              dict(firstname='Jane', lastname='Doe', grade=4),
...              dict(firstname='John', lastname='Doe', grade=3)]
```

Let's start by looking at an example without selector factories. Our query will be to order the employees by descending grade, then by ascending last name and finally by ascending first name:

```
>>>  query(employees).order_by_descending(lambda employee: employee['grade']) \
...                  .then_by(lambda employee: employee['lastname'])          \
...                  .then_by(lambda employee: employee['firstname']).to_list()
[{'grade': 4, 'lastname': 'Doe', 'firstname': 'Jane'},
 {'grade': 3, 'lastname': 'Bloggs', 'firstname': 'Joe'},
 {'grade': 3, 'lastname': 'Doe', 'firstname': 'John'},
 {'grade': 3, 'lastname': 'Nordmann', 'firstname': 'Ola'},
 {'grade': 2, 'lastname': 'Nordmann', 'firstname': 'Kari'}]
```

Those lambda expressions can be a bit of a mouthful, especially given Python's less-than-concise lambda syntax. We can improve by using less descriptive names for the lambda arguments:

```
>>>  query(employees).order_by_descending(lambda e: e['grade'])  \
...                  .then_by(lambda e: e['lastname'])           \
...                  .then_by(lambda e: e['firstname']).to_list()
[{'grade': 4, 'lastname': 'Doe', 'firstname': 'Jane'},
 {'grade': 3, 'lastname': 'Bloggs', 'firstname': 'Joe'},
 {'grade': 3, 'lastname': 'Doe', 'firstname': 'John'},
 {'grade': 3, 'lastname': 'Nordmann', 'firstname': 'Ola'},
 {'grade': 2, 'lastname': 'Nordmann', 'firstname': 'Kari'}]
```

but there's still quite a lot of syntactic noise in here. By using one of the selector factories provided by `asq` we can make this example more concise. The particular selector factory we are going to use is called *k_()* where the *k* is a mnemonic for 'key' and the underscore is there purely to make the name more unusual to avoid consuming a useful single letter variable name from the importing namespace. `k_()` takes a single argument which is the name of the key to be used when indexing into the element, so the expressions:

```
k_('foo')
```

and:

```
lambda x: x['foo']
```

are equivalent because in fact the first expression is in fact returning the second one. Let's see `k_()` in action reducing the verbosity and apparent complexity of the query somewhat:

```
>>> from asq import k_
>>> query(employees).order_by_descending(k_('grade'))   \
...                 .then_by(k_('lastname'))            \
...                 .then_by(k_('firstname')).to_list()
[{'grade': 4, 'lastname': 'Doe', 'firstname': 'Jane'},
 {'grade': 3, 'lastname': 'Bloggs', 'firstname': 'Joe'},
 {'grade': 3, 'lastname': 'Doe', 'firstname': 'John'},
 {'grade': 3, 'lastname': 'Nordmann', 'firstname': 'Ola'},
 {'grade': 2, 'lastname': 'Nordmann', 'firstname': 'Kari'}]
```

It might not be immediately obvious from it's name, but `k_()` works with any object supporting indexing with square brackets, so it can also be used with an integer 'key' for retrieved results from sequences such as lists and tuples.

Attribute selector factory

The attribute selector factory provided by `asq` is called *a_()* and it creates a selector which retrieves a named attribute from each element. To illustrate its utility, we'll re-run the key selector exercise using the attribute selector against `Employee` objects rather than dictionaries. First of all, our `Employee` class:

```
>>> class Employee(object):
...     def __init__(self, firstname, lastname, grade):
...         self.firstname = firstname
...         self.lastname = lastname
...         self.grade = grade
...     def __repr__(self):
...         return ("Employee(" + repr(self.firstname) + ", "
...                             + repr(self.lastname) + ", "
...                             + repr(self.grade) + ")")
```

Now the query and its result use the lambda form for the selectors:

```
>>> query(employees).order_by_descending(lambda employee: employee.grade)  \
...                  .then_by(lambda employee: employee.lastname)           \
...                  .then_by(lambda employee: employee.firstname).to_list()
[Employee('Jane', 'Doe', 4), Employee('Joe', 'Bloggs', 3),
 Employee('John', 'Doe', 3), Employee('Ola', 'Nordmann', 3),
 Employee('Kari', 'Nordmann', 2)]
```

We can make this query more concise by creating our selectors using the `a_` selector factory, where the *a* is a mnemonic for 'attribute'. `a_()` accepts a single argument which is the name of the attribute to get from each element. The expression:

```
a_('foo')
```

is equivalent to:

```
lambda x: x.foo
```

Using this construct we can shorted our query to the more concise:

```
>>> query(employees).order_by_descending(a_('grade'))  \
...                  .then_by(a_('lastname'))           \
...                  .then_by(a_('firstname')).to_list()
[Employee('Jane', 'Doe', 4), Employee('Joe', 'Bloggs', 3),
 Employee('John', 'Doe', 3), Employee('Ola', 'Nordmann', 3),
 Employee('Kari', 'Nordmann', 2)]
```

Method selector factory

The method-call selector factory provided by `asq` is called *m_()* and it creates a selector which makes a method call on each element, optionally passing positional or named arguments to the method. We'll re-run the attribute selector exercise using the method selector against a modified `Employee` class which incorporates a couple of methods:

```
>>> class Employee(object):
...     def __init__(self, firstname, lastname, grade):
...         self.firstname = firstname
...         self.lastname = lastname
...         self.grade = grade
...     def __repr__(self):
...         return ("Employee(" + repr(self.firstname)
...                             + repr(self.lastname)
...                             + repr(self.grade) + ")")
...     def full_name(self):
...         return self.firstname + " " + self.lastname
...     def award_bonus(self, base_amount):
...         return self.grade * base_amount
```

In its simplest form, the `m_()` selector factory takes a single argument, which is the name of the method to be called as a string. So:

```
m_('foo')
```

is equivalent to:

```
lambda x: x.foo()
```

We can use this to easy generate a list of full names for our employees:

```
>>> query(employees).select(m_('full_name')).to_list()
['Joe Bloggs', 'Ola Nordmann', 'Kari Nordmann', 'Jane Doe', 'John Doe']
```

The `m_()` selector factory also accepts arbitrary number of additional positional or named arguments which will be forwarded to the method when it is called on each element. So:

```
m_('foo', 42)
```

is equivalent to:

```
lambda x: x.foo(42)
```

For example to determine total cost of awarding bonuses to our employees on the basis of grade, we can do:

```
>>> query(employees).select(m_('award_bonus', 1000)).to_list()
[3000, 3000, 2000, 4000, 3000]
```

Default selectors and the identity selector

Any of the selector arguments to query operators in `asq` may be omitted[1] to allow the use of operators to be simplified. When a selector is omitted the default is used and the documentation makes it clear how that default behaves. In most cases, the default selector is the `identity()` selector. The identity selector is very simple and is equivalent to:

```
def identity(x):
    return x
```

That is, it is a function that returns it's only argument - essentially it's a do-nothing function. This is useful because frequently we don't want to select an attribute or key from an element - we want to use the element value directly. For example, to sort a list of words alphabetically, we can omit the selector passed to `order_by()` allowing if to default to the identity selector:

```
>>> words = "the quick brown fox jumped over the lazy dog".split()
>>> query(words).order_by().to_list()
['brown', 'dog', 'fox', 'jumped', 'lazy', 'over', 'quick', 'the', 'the']
```

Some query operators, notably `select()` perform important optimisations when used with the identity operator. For example the operator `select(identity)` does nothing and simply returns the Queryable on which it was invoked.

Predicates

Many of the query operators, such as `where()`, `distinct()`, `skip()`, accept predicates. Predicates are functions which return `True` or `False`. As with selectors (see above) predicates can be defined with lambdas, functions, unbound methods, bound methods or indeed any other callable that returns True or False. For convenience `asq` also provides some predicate factories and combinators to concisely build predicates for common situations.

Lambdas

[1] Except the single selector argument to the `select()` operator itself.

```
>>> numbers = [0, 56, 23, 78, 94, 56, 12, 34, 36, 90, 23, 76, 4, 67]
>>> query(numbers).where(lambda x: x > 35).to_list()
[56, 78, 94, 56, 36, 90, 76, 67]
```

Functions

Here we use the `bool()` built-in function to remove zeros from the list:

```
>>> numbers = [0, 56, 23, 78, 94, 56, 12, 34, 36, 90, 23, 76, 4, 67]
>>> query(numbers).where(bool).to_list()
[56, 23, 78, 94, 56, 12, 34, 36, 90, 23, 76, 4, 67]
```

Unbound methods

Here we use an unbound method of the `str` class to extract only alphabetic strings from a list:

```
>>> a = ['zero', 'one', '2', '3', 'four', 'five', '6', 'seven', 'eight', '9']
>>> query(a).where(str.isalpha).to_list()
['zero', 'one', 'four', 'five', 'seven', 'eight']
```

Bound methods

Bound methods are obtained by referencing the method of an *instance* rather than the method of a class. That is, the instance referred to by the *self* parameter passed as the first argument of a method has already been determined.

To illustrate, here we create a variation of Multiplier class earlier with a method to test whether a given number is a multiple of the supplied factor:

```
>>> numbers = [1, 18, 34, 23, 56, 48, 45]
>>>
>>> class Multiplier(object):
...     def __init__(self, factor):
...         self.factor = factor
...     def is_multiple(self, value):
...         return value % self.factor == 0
...
>>> six_multiplier = Multiplier(6)
>>>
>>> is_six_a_factor = six_multiplier.is_multiple
>>> is_six_a_factor
<bound method Multiplier.is_multiple of <__main__.Multiplier object at 0x029FEDF0>>
>>>
>>> query(numbers).where(is_six_a_factor).to_list()
[18, 48]
```

This has the effect of passing each element of the sequence in turn as an argument to the bound method which returns True or False.

Predicate factories

For complex predicates inline lambdas can become quite verbose and have limited readability. To mitigate this somewhat, `asq` provides some predicate factories and predicate combinators.

The provided predicates are:

Predicate factory	Created selector function
`eq_(value)`	`lambda x: x == value`
`ne_(value)`	`lambda x: x != value`
`lt_(value)`	`lambda x: x < value`
`le_(value)`	`lambda x: x <= value`
`ge_(value)`	`lambda x: x >= value`
`gt_(value)`	`lambda x: x >= value`
`is_(value)`	`lambda x: x is value`
`contains_(value)`	`lambda x: value in x`

Predicates are available in the `predicates` module of the `asq` package:

```
>>> from asq.predicates import *
```

So given:

```
>>> numbers = [0, 56, 23, 78, 94, 56, 12, 34, 36, 90, 23, 76, 4, 67]
```

the query expression:

```
>>> query(numbers).where(lambda x: x > 35).take_while(lambda x: x < 90).to_list()
[56, 78]
```

could be written more succinctly rendered as:

```
>>> query(numbers).where(gt_(35)).take_while(lt_(90)).to_list()
[56, 78]
```

Predicate combinator factories

Some simple combinators are provided to allow the predicate factories to be combined to form more powerful expressions. These combinators are,

Combinator factory	Created selector function
`not_(a)`	`lambda x: not a(x)`
`and_(a, b)`	`lambda x: a(x) and b(x)`
`or_(a, b)`	`lambda x: a(x) or b(x)`
`xor(a, b)`	`lambda x: a(x) != b(x)`

where `a` and `b` are themselves predicates.

So given:

```
>>> numbers = [0, 56, 23, 78, 94, 56, 12, 34, 36, 90, 23, 76, 4, 67]
```

the query expression:

```
>>> query(numbers).where(lambda x: x > 20 and x < 80).to_list()
[56, 23, 78, 56, 34, 36, 23, 76, 67]
```

could be expressed as:

```
>>> query(numbers).where(and_(gt_(20), lt_(80).to_list()
[56, 23, 78, 56, 34, 36, 23, 76, 67]
```

Although complex expressions are probably still better expressed as lambdas or separate functions altogether.

Using selector factories for predicates

A predicate is any callable that returns `True` or `False`, so any selector which returns `True` or `False` is by definition a predicate. This means that the selector factories `k_()`, `a_()` and `m_()` may also be used as predicate factories so long as they return boolean values. They may also be used with the predicate combinators. For example, consider a sequence of `Employee` objects which have an `intern` attribute which evaluates to True or False. We can filter out interns using this query:

```
>>> query(employees).where(not_(a_('intern')))
```

Comparers

Some of the query operators accept equality comparers. Equality comparers are callables which can be used to determine whether two value should be considered equal for the purposes of a query. For example, the `contains()` query operator accepts an optional equality comparer used for determining membership. To illustrate, we will use the `insensitive_eq()` comparer which does a case insensitive equality test:

```
>>> from asq.comparers import insensitive_eq
>>> names = ['Matthew', 'Mark', 'John']
>>> query(names).contains('MARK', insensitive_eq)
True
```

Records

In all of the examples in this documentation so far, the data to be queried has either been represented as combinations of built-in Python types such as lists and dictionaries, or we have needed define specific classes to represented our data. Sometimes there's a need for a type without the syntactic clutter of say dictionaries, but without the overhead of creating a whole class with methods; you just want to bunch some data together. The `Record` type provided by `asq` fulfills this need. A convenience function called `new()` can be used to concisely create Records. To use new, just pass in named arguments to define the Record properties:

```
>>> product = new(id=5723, name="Mouse", price=33, total_revenue=23212)
>>> product
Record(id=5723, price=33, total_revenue=23212, name='Mouse')
```

And retrieve properties using regular Python attribute syntax:

```
>>> product.price
33
```

This can be useful when we want to carry several derived values through a query such as in this example where we create Records containing the full names and highest score of students, we then sort the records by the high score:

```
>>> from pupils import students
>>> students
[{'lastname': 'Blogs', 'firstname': 'Joe', 'scores': [56, 23, 21, 89]},
```

```
 {'lastname': 'Doe', 'firstname': 'John', 'scores': [34, 12, 92, 93]},
 {'lastname': 'Doe', 'firstname': 'Jane', 'scores': [33, 94, 91, 13]},
 {'lastname': 'Nordmann', 'firstname': 'Ola', 'scores': [98, 23, 98, 87]},
 {'lastname': 'Nordmann', 'firstname': 'Kari', 'scores': [86, 37, 88, 87]},
 {'lastname': 'Rossi', 'firstname': 'Mario', 'scores': [37, 95, 45, 18]}]
>>> query(students).select(lambda s: new(name="{firstname} {lastname}".format(**s),
...                                      high_score=max(s['scores']))) \
...                .order_by(a_('high_score').to_list()
[Record(high_score=88, name='Kari Nordmann'),
 Record(high_score=89, name='Joe Blogs'),
 Record(high_score=93, name='John Doe'),
 Record(high_score=94, name='Jane Doe'),
 Record(high_score=95, name='Mario Rossi'),
 Record(high_score=98, name='Ola Nordmann')]
```

Debugging

With potentially so much deferred execution occurring, debugging `asq` query expressions using tools such as debuggers can be challenging. Furthermore, since queries are expressions use of statements such as Python 2 `print` can be awkward.

To ease debugging, `asq` provides a logging facility which can be used to display intermediate results with an optional ability for force full, rather than lazy, evaluation of sequences.

To demonstrate, let's start with a bug-ridden implementation of Fizz-Buzz implemented with `asq`. Fizz-Buzz is a game where the numbers 1 to 100 are read aloud but for numbers divisible by three "Fizz" is shouted, and for numbers divisible by five, "Buzz" is shouted.

```
>>> from asq.initiators import integers
>>> integers(1, 100).select(lambda x: "Fizz" if x % 3 == 0 else x) \
...                 .select(lambda x: "Buzz" if x % 5 == 0 else x).to_list()
```

At a glance this looks like it should work, but when run we get:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "asq/queryables.py", line 1910, in to_list
    lst = list(self)
  File "<stdin>", line 1, in <lambda>
TypeError: not all arguments converted during string formatting
```

To investigate further it would be useful to examine the intermediate results. We can do this using the `log()` query operator, which accepts any logger supporting a `debug(message)` method. We can get just such a logger from the Python standard library `logging` module:

```
>>> import logging
>>> clog = logging.getLogger("clog")
>>> clog.setLevel(logging.DEBUG)
```

which creates a console logger we have called `clog`:

```
>>> from asq.initiators import integers
>>> integers(1, 100) \
...  .select(lambda x: "Fizz" if x % 3 == 0 else x).log(clog, label="Fizz select"). \
...  .select(lambda x: "Buzz" if x % 5 == 0 else x).to_list()
DEBUG:clog:Fizz select : BEGIN (DEFERRED)
```

```
DEBUG:clog:Fizz select : [0] yields 1
DEBUG:clog:Fizz select : [1] yields 2
DEBUG:clog:Fizz select : [2] yields 'Fizz'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "asq/queryables.py", line 1910, in to_list
    lst = list(self)
  File "<stdin>", line 1, in <lambda>
TypeError: not all arguments converted during string formatting
```

so we can see the the first select operator yields 1, 2, 'Fizz' before the failure. Now it's perhaps more obvious that when x in the second lambda is equal to 'Fizz' the `%` operator will be operating on a string on its left-hand side and so the ` % will perform string interpolation rather than modulus. This is the cause of the error we see.

We can fix this by not applying the modulus operator in the case that x is 'Fizz':

```
>>> integers(1, 100).select(lambda x: "Fizz" if x % 3 == 0 else x).log(clog, label=
↪"Fizz select") \
                    .select(lambda x: "Buzz" if x != "Fizz" and x % 5 == 0 else x).to_
↪list()
DEBUG:clog:Fizz select : BEGIN (DEFERRED)
DEBUG:clog:Fizz select : [0] yields 1
DEBUG:clog:Fizz select : [1] yields 2
DEBUG:clog:Fizz select : [2] yields 'Fizz'
DEBUG:clog:Fizz select : [3] yields 4
DEBUG:clog:Fizz select : [4] yields 5
DEBUG:clog:Fizz select : [5] yields 'Fizz'
DEBUG:clog:Fizz select : [6] yields 7
DEBUG:clog:Fizz select : [7] yields 8
DEBUG:clog:Fizz select : [8] yields 'Fizz'
DEBUG:clog:Fizz select : [9] yields 10
DEBUG:clog:Fizz select : [10] yields 11
DEBUG:clog:Fizz select : [11] yields 'Fizz'
DEBUG:clog:Fizz select : [12] yields 13
DEBUG:clog:Fizz select : [13] yields 14
DEBUG:clog:Fizz select : [14] yields 'Fizz'
DEBUG:clog:Fizz select : [15] yields 16
DEBUG:clog:Fizz select : [16] yields 17
...
DEBUG:clog2:Fizz select : [98] yields 'Fizz'
DEBUG:clog2:Fizz select : [99] yields 100
DEBUG:clog2:Fizz select : END (DEFERRED)
[1, 2, 'Fizz', 4, 'Buzz', 'Fizz', 7, 8, 'Fizz', 'Buzz', 11, 'Fizz', 13, 14,
 'Fizz', 16, 17, 'Fizz', 19, 'Buzz', 'Fizz', 22, 23, 'Fizz', 'Buzz', 26,
 'Fizz', 28, 29, 'Fizz', 31, 32, 'Fizz', 34, 'Buzz', 'Fizz', 37, 38, 'Fizz',
 'Buzz', 41, 'Fizz', 43, 44, 'Fizz', 46, 47, 'Fizz', 49, 'Buzz', 'Fizz', 52,
 53, 'Fizz', 'Buzz', 56, 'Fizz', 58, 59, 'Fizz', 61, 62, 'Fizz', 64, 'Buzz',
 'Fizz', 67, 68, 'Fizz', 'Buzz', 71, 'Fizz', 73, 74, 'Fizz', 76, 77, 'Fizz',
 79, 'Buzz', 'Fizz', 82, 83, 'Fizz', 'Buzz', 86, 'Fizz', 88, 89, 'Fizz', 91,
 92, 'Fizz', 94, 'Buzz', 'Fizz', 97, 98, 'Fizz', 'Buzz']
```

That problem is solved, but inspection of the output shows that our query expression produces incorrect results for those numbers which are multiples of both 3 and 5, such as 15, for which we should be returning 'FizzBuzz'. For the sake of completeness, let's modify the expression to deal with this:

```
>>> integers(1, 100).select(lambda x: "FizzBuzz" if x % 15 == 0 else x) \
                    .select(lambda x: "Fizz" if x != "FizzBuzz" and x % 3 == 0 else␣
↪x) \
```

```
                    .select(lambda x: "Buzz" if x != "FizzBuzz" and x != "Fizz" and x
↪% 5 == 0 else x).to_list()
[1, 2, 'Fizz', 4, 'Buzz', 'Fizz', 7, 8, 'Fizz', 'Buzz', 11, 'Fizz', 13, 14,
 'FizzBuzz', 16, 17, 'Fizz', 19, 'Buzz', 'Fizz', 22, 23, 'Fizz', 'Buzz', 26,
 'Fizz', 28, 29, 'FizzBuzz', 31, 32, 'Fizz', 34, 'Buzz', 'Fizz', 37, 38,
 'Fizz', 'Buzz', 41, 'Fizz', 43, 44, 'FizzBuzz', 46, 47, 'Fizz', 49, 'Buzz',
 'Fizz', 52, 53, 'Fizz', 'Buzz', 56, 'Fizz', 58, 59, 'FizzBuzz', 61, 62,
 'Fizz', 64, 'Buzz', 'Fizz', 67, 68, 'Fizz', 'Buzz', 71, 'Fizz', 73, 74,
 'FizzBuzz', 76, 77, 'Fizz', 79, 'Buzz', 'Fizz', 82, 83, 'Fizz', 'Buzz', 86,
 'Fizz', 88, 89, 'FizzBuzz', 91, 92, 'Fizz', 94, 'Buzz', 'Fizz', 97, 98,
 'Fizz', 'Buzz']
```

Extending `asq`

For .NET developers

The @extend decorator described here performs the same role as C# extension methods to IEnumerable play in Microsoft's LINQ.

The fluent interface of `asq` works by chaining method calls on Queryable types, so to extend `asq` with new query operators must be able to add methods to Queryable. New methods added in this way must have a particular structure in order to be usable in the middle of a query chain.

To define a new query operator, use the @extend function decorator from the `asq.extension` package to decorator a module scope function. To illustrate, let's add a new operator which adds a separating item between existing items:

```
@extend(Queryable)
def separate_with(self, separator):
    '''Insert a separator between items.

    Note: This method uses deferred execution.

    Args:
        separator: The separating element to be inserted between each source
            element.

    Returns:
        A Queryable over the separated sequence.
    '''

    # Validate the arguments.  It is important to validate the arguments
    # eagerly, when the operator called, rather than when the result is
    # evaluated to ease debugging.
    if self.closed():
        raise ValueError("Attempt to call separate_with() on a closed Queryable.")

    # In order to get deferred execution (lazy evaluation) we need to define
    # a generator. This generator is also a closure over the parameters to
    # separate_with, namely 'self' and 'separator'.
    def generator():
        # Create an iterator over the source sequence - self is a Queryable
        # which is iterable.
        i = iter(self)

        # Attempt to yield the first element, which may or may not exist;
```

```
        # next() will raise StopIteration if it does not, so we exit.
        try:
            yield next(i)
        except StopIteration:
            return

        # Alternately yield a separator and the next element for all
        # remaining elements in the source sequence.
        for item in i:
            yield separator
            yield item

    # Create a new Queryable from the generator, by calling the _create()
    # factory function, rather than by calling the Queryable constructor
    # directly.  This ensures that the correct subclass of Queryable is
    # created.
    return self._create(generator())
```

The @extend decorator installs the new operator so it may be used immediately:

```
a = [1, 4, 9, 2, 3]
query(a).select(lambda x: x*x).separate_with(0).to_list()
```

which gives:

```
[1, 0, 16, 0, 81, 0, 4, 0, 9]
```

1.3 Reference Documentation

Descriptions and examples for every public function, class and method in `asq`.

1.3.1 API Reference

asq

asq.initiators

Initiators are factory functions for creating Queryables.

Initiators are so-called because they are used to initiate a query expression using the fluent interface of `asq` which uses method-chaining to compose complex queries from the query operators provided by queryables.

query	Make an iterable queryable.
empty	An empty Queryable.
integers	Generates in sequence the integral numbers within a range.
repeat	Generate a sequence with one repeated value.

asq.initiators.**query**(*iterable*)
Make an iterable queryable.

Use this function as an entry-point to the asq system of chainable query methods.

Note: Currently this factory only provides support for objects supporting the iterator protocol. Future implementations may support other providers.

Parameters **iterable** – Any object supporting the iterator protocol.

Returns An instance of Queryable.

Raises TypeError - If iterable is not actually iterable

Examples

Create a queryable from a list:

```
>>> from asq.initiators import query
>>> a = [1, 7, 9, 4, 3, 2]
>>> q = query(a)
>>> q
Queryable([1, 7, 9, 4, 3, 2])
>>> q.to_list()
[1, 7, 9, 4, 3, 2]
```

asq.initiators.**empty**()
An empty Queryable.

Note: The same empty instance will be returned each time.

Returns A Queryable over an empty sequence.

Examples

Create a queryable from a list:

```
>>> from asq.initiators import empty
>>> q = empty()
>>> q
Queryable(())
>>> q.to_list()
[]
```

See that `empty()` always returns the same instance:

```
>>> a = empty()
>>> b = empty()
>>> a is b
True
```

asq.initiators.**integers**(*start*, *count*)
Generates in sequence the integral numbers within a range.

Note: This method uses deferred execution.

Parameters

- **start** – The first integer in the sequence.
- **count** – The number of sequential integers to generate.

Returns A Queryable over the specified range of integers.

Raises ValueError - If count is negative.

Examples

Create the first five integers:

```
>>> from asq.initiators import integers
>>> numbers = integers(0, 5)
>>> numbers
Queryable(range(0, 5))
>>> numbers.to_list()
[0, 1, 2, 3, 4]
```

asq.initiators.**repeat**(*element*, *count*)
Generate a sequence with one repeated value.

Note: This method uses deferred execution.

Parameters

- **element** – The value to be repeated.
- **count** – The number of times to repeat the value.

Raises ValueError - If the count is negative.

Examples

Repeat the letter x five times:

```
>>> from asq.initiators import repeat
>>> q = repeat('x', 5)
>>> q
Queryable(repeat('x', 5))
>>> q.to_list()
['x', 'x', 'x', 'x', 'x']
```

asq.queryables

Classes which support the Queryable interface.

asq.queryables.Queryable

class asq.queryables.**Queryable**(*iterable*)
Queries over iterables executed serially.

Queryable objects are constructed from iterables.

Queryable.__contains__	Support for membership testing using the 'in' operator.
Queryable.__enter__	Support for the context manager protocol.
Queryable.__eq__	Determine value equality with another iterable.
Queryable.__exit__	Support for the context manager protocol.
Queryable.__getitem__	Support for indexing into the sequence using square brackets.
Queryable.__init__	Construct a Queryable from any iterable.
Queryable.__iter__	Support for the iterator protocol.
Queryable.__ne__	Determine value inequality with another iterable.
Queryable.__reversed__	Support for sequence reversal using the reversed() built-in.
Queryable.__repr__	Returns a stringified representation of the Queryable.
Queryable.__str__	Returns a stringified representation of the Queryable.
Queryable.aggregate	Apply a function over a sequence to produce a single result.
Queryable.all	Determine if all elements in the source sequence satisfy a condition.
Queryable.any	Determine if the source sequence contains any elements which satisfy the predicate.
Queryable.as_parallel	Return a ParallelQueryable for parallel execution of queries.
Queryable.average	Return the arithmetic mean of the values in the sequence..
Queryable.close	Closes the queryable.
Queryable.closed	Determine whether the Queryable has been closed.
Queryable.concat	Concatenates two sequences.
Queryable.contains	Determines whether the sequence contains a particular value.
Queryable.count	Return the number of elements (which match an optional predicate).
Queryable.default_if_empty	If the source sequence is empty return a single element sequence containing the supplied default value, otherwise return the source sequence unchanged.
Queryable.difference	Returns those elements which are in the source sequence which are not in the second_iterable.
Queryable.distinct	Eliminate duplicate elements from a sequence.
Queryable.element_at	Return the element at ordinal index.
Queryable.first	The first element in a sequence (optionally satisfying a predicate).
Queryable.first_or_default	The first element (optionally satisfying a predicate) or a default.
Queryable.group_by	Groups the elements according to the value of a key extracted by a selector function.
Queryable.group_join	Match elements of two sequences using keys and group the results.

Continued on next page

Table 1.2 – continued from previous page

Queryable.intersect	Returns those elements which are both in the source sequence and in the second_iterable.
Queryable.join	Perform an inner join with a second sequence using selected keys.
Queryable.last	The last element in a sequence (optionally satisfying a predicate).
Queryable.last_or_default	The last element (optionally satisfying a predicate) or a default.
Queryable.log	Log query result consumption details to a logger.
Queryable.max	Return the maximum value in a sequence.
Queryable.min	Return the minimum value in a sequence.
Queryable.of_type	Filters elements according to whether they are of a certain type.
Queryable.order_by	Sorts by a key in ascending order.
Queryable.order_by_descending	Sorts by a key in descending order.
Queryable.select	Transforms each element of a sequence into a new form.
Queryable.select_many	Projects each element of a sequence to an intermediate new sequence, flattens the resulting sequences into one sequence and optionally transforms the flattened sequence using a selector function.
Queryable.select_many_with_correspondence	Projects each element of a sequence to an intermediate new sequence, and flattens the resulting sequence, into one sequence and uses a selector function to incorporate the corresponding source for each item in the result sequence.
Queryable.select_many_with_index	Projects each element of a sequence to an intermediate new sequence, incorporating the index of the element, flattens the resulting sequence into one sequence and optionally transforms the flattened sequence using a selector function.
Queryable.select_with_correspondence	Apply a callable to each element in an input sequence, generating a new sequence of 2-tuples where the first element is the input value and the second is the transformed input value.
Queryable.select_with_index	Transforms each element of a sequence into a new form, incorporating the index of the element.
Queryable.sequence_equal	Determine whether two sequences are equal by elementwise comparison.
Queryable.single	The only element (which satisfies a condition).
Queryable.single_or_default	The only element (which satisfies a condition) or a default.
Queryable.skip	Skip the first count contiguous elements of the source sequence.
Queryable.skip_while	Omit elements from the start for which a predicate is True.
Queryable.sum	Return the arithmetic sum of the values in the sequence..
Queryable.take	Returns a specified number of elements from the start of a sequence.

Continued on next page

Table 1.2 – continued from previous page

Queryable.take_while	Returns elements from the start while the predicate is True.
Queryable.to_dictionary	Build a dictionary from the source sequence.
Queryable.to_list	Convert the source sequence to a list.
Queryable.to_lookup	Returns a Lookup object, using the provided selector to generate a key for each item.
Queryable.to_set	Convert the source sequence to a set.
Queryable.to_str	Build a string from the source sequence.
Queryable.to_tuple	Convert the source sequence to a tuple.
Queryable.union	Returns those elements which are either in the source sequence or in the second_iterable, or in both.
Queryable.where	Filters elements according to whether they match a predicate.
Queryable.zip	Elementwise combination of two sequences.

__contains__(*item*)

Support for membership testing using the 'in' operator.

Parameters **item** – The item for which to test membership.

Returns True if item is in the sequence, otherwise False.

Note: A chainable query operator called `contains()` (no underscores) is also provided.

Example

Test whether 49 is one of the squares of two, seven or nine:

```
>>> a = [2, 7, 9]
>>> 49 in query(a).select(lambda x: x*x)
True
```

__enter__()

Support for the context manager protocol.

__eq__(*rhs*)

Determine value equality with another iterable.

Parameters **rhs** – Any iterable collection.

Returns True if the sequences are equal in value, otherwise False.

Note: This in the infix operator equivalent of the sequence_equal() query operator.

Examples

Test whether a sequence is equal to a list:

```
>>> expected = [2, 4, 8, 16, 32]
>>> range(1, 5).select(lambda x: x ** 2) == expected
True
```

__exit__(*type*, *value*, *traceback*)
Support for the context manager protocol.

Ensures that close() is called on the Queryable.

__getitem__(*index*)
Support for indexing into the sequence using square brackets.

Equivalent to element_at().

Parameters **index** – The index should be between zero and count() - 1 inclusive. Negative indices are not interpreted in the same way they are for built-in lists, and are considered out-of-range.

Returns The value of the element at offset index into the sequence.

Raises

- ValueError - If the Queryable is closed().
- IndexError - If the index is out-of-range.

Note: A chainable query operator called `element_at()` is also provided.

Examples

Retrieve the fourth element of a greater than six:

```
>>> a = [7, 3, 9, 2, 1, 10, 11, 4, 13]
>>> query(a).where(lambda x: x > 6)[3]
11
```

__init__(*iterable*)
Construct a Queryable from any iterable.

Parameters **iterable** – Any object supporting the iterator protocol.

Raises TypeError - if iterable does not support the iterator protocol.

Example

Initialise a queryable from a list:

```
>>> a = [1, 5, 7, 8]
>>> queryable = Queryable(a)
```

Note: The `query(iterable)` initiator should normally be used in preference to calling the `Queryable` constructor directly.

__iter__()
Support for the iterator protocol.

Allows Queryable instances to be used anywhere an iterable is required.

Returns An iterator over the values in the query result.

Raises ValueError - If the Queryable has been closed().

Note: This method should not usually be called directly; use the `iter()` built-in or other Python constructs which check for the presence of `__iter__()`, such as `for` loops.

Examples

Call `__iter__()` indirectly through the `iter()` built-in to obtain an iterator over the query results:

```
>>> a = [8, 9, 2]
>>> q = query(a)
>>> iterator = iter(q)
>>> next(iterator)
8
>>> next(iterator)
9
>>> next(iterator)
2
>>> next(iterator)
StopIteration
```

Call `__iter__()` indirectly by using a `for` loop:

```
>>> a = [1, 9, 4]
>>> q = query(a)
>>> for v in q:
...     print(v)
...
1
9
4
```

__ne__(*rhs*)

Determine value inequality with another iterable.

Parameters **rhs** – Any iterable collection.

Returns True if the sequences are inequal in value, otherwise False.

Examples

Test whether a sequence is not equal to a list:

```
>>> expected = [1, 2, 3]
>>> range(1, 5).select(lambda x: x ** 2) != expected
True
```

__reversed__()

Support for sequence reversal using the reversed() built-in.

Called by reversed() to implement reverse iteration.

Equivalent to the reverse() method.

Returns A Queryable over the reversed sequence.

Raises ValueError - If the Queryable is closed().

Note: A chainable query operator called `reverse()` is also provided.

Note: This method should not usually be called directly; use the `reversed()` built-in or other Python constructs which check for the presence of `__reversed__()`.

Example

Create a reverse iterator over a queryable for use with a `for` loop:

```
>>> a = [7, 3, 9, 2, 1]
>>> q = query(a)
>>> for v in reversed(q):
...     print(v)
...
1
2
9
3
7
```

__repr__()
Returns a stringified representation of the Queryable.

The string will *not* necessarily contain the sequence data.

Returns A stringified representation of the Queryable.

Note: This method should not usually be called directly; use the `str()` built-in or other Python constructs which check for the presence of __str__ such as string interpolation functions.

Provide a string representation of the Queryable using the `repr()` built-in:

```
>>> a = [9, 7, 8]
>>> q = query(a)
>>> str(q)
'Queryable([9, 7, 8])'
```

__str__()
Returns a stringified representation of the Queryable.

The string *will* necessarily contain the sequence data.

Returns A stringified representation of the Queryable.

Note: This method should not usually be called directly; use the `str()` built-in or other Python constructs which check for the presence of __str__ such as string interpolation functions.

Note: In order to convert the Queryable sequence to a string based on the element values, consider using the `to_str()` method.

Example

Convert the Queryable to a string using the `str()` built-in:

```
>>> a = [9, 7, 8]
>>> q = query(a)
>>> str(q)
'Queryable([9, 7, 8])'
```

aggregate(*reducer*, *seed=sentinel*, *result_selector=identity*)
Apply a function over a sequence to produce a single result.

Apply a binary function cumulatively to the elements of the source sequence so as to reduce the iterable to a single value.

Note: This method uses immediate execution.

Parameters

- **reducer** – A binary function the first positional argument of which is an accumulated value and the second is the update value from the source sequence. The return value should be the new accumulated value after the update value has been incorporated.
- **seed** – An optional value used to initialise the accumulator before iteration over the source sequence. If seed is omitted the and the source sequence contains only one item, then that item is returned.
- **result_selector** – An optional unary function applied to the final accumulator value to produce the result. If omitted, defaults to the identity function.

Raises

- ValueError - If called on an empty sequence with no seed value.
- TypeError - If reducer is not callable.
- TypeError - If result_selector is not callable.

Examples

Compute the product of a list of numbers:

```
>>> numbers = [4, 7, 3, 2, 1, 9]
>>> query(numbers).aggregate(lambda accumulator, update: accumulator␣
↪* update)
1512
```

Concatenate strings to an initial seed value:

```
>>> cheeses = ['Cheddar', 'Stilton', 'Cheshire', 'Beaufort', 'Brie']
>>> query(cheeses).aggregate(lambda a, u: a + ' ' + u, seed="Cheeses:
↪")
'Cheeses: Cheddar Stilton Cheshire Beaufort Brie'
```

Concatenate text fragments using `operator.add()` and return the number of words:

```
>>> from operator import add
>>> fragments = ['The quick ', 'brown ', 'fox jumped over ', 'the ',
↪'lazy dog.']
>>> query(fragments).aggregate(add, lambda result: len(result.
↪split()))
9
```

all(*predicate=bool*)
Determine if all elements in the source sequence satisfy a condition.

All of the source sequence will be consumed.

Note: This method uses immediate execution.

Parameters **predicate** (*callable*) – An optional single argument function used to test each elements. If omitted, the bool() function is used resulting in the elements being tested directly.

Returns True if all elements in the sequence meet the predicate condition, otherwise False.

Raises

- ValueError - If the Queryable is closed()
- TypeError - If predicate is not callable.

Examples

Determine whether all values evaluate to True in a boolean context:

```
>>> items = [5, 2, "camel", 3.142, (3, 4, 9)]
>>> query(objects).all()
True
```

Check that all numbers are divisible by 13:

```
>>> numbers = [260, 273, 286, 299, 312, 325, 338, 351, 364, 377]
>>> query(numbers).all(lambda x: x % 13 == 0)
True
```

any (*predicate=None*)

Determine if the source sequence contains any elements which satisfy the predicate.

Only enough of the sequence to satisfy the predicate once is consumed.

Note: This method uses immediate execution.

Parameters **predicate** – An optional single argument function used to test each element. If omitted, or None, this method returns True if there is at least one element in the source.

Returns True if the sequence contains at least one element which satisfies the predicate, otherwise False.

Raises ValueError - If the Queryable is closed()

Examples

Determine whether the sequence contains any items:

```
>>> items = [0, 0, 0]
>>> query(items).any()
True
```

Determine whether the sequence contains any items which are a multiple of 13:

```
>>> numbers = [98, 458, 32, 876, 12, 9, 325]
>>> query(numbers).any(lambda x: x % 13 == 0)
True
```

as_parallel(*pool=None*)
Return a ParallelQueryable for parallel execution of queries.

> **Warning:** This feature should be considered experimental alpha quality.

Parameters **pool** – An optional multiprocessing pool which will provide execution resources for parellel processing. If omitted, a pool will be created if necessary and managed internally.
Returns A ParallelQueryable on which all the standard query operators may be called.

average(*selector=identity*)
Return the arithmetic mean of the values in the sequence..

All of the source sequence will be consumed.

Note: This method uses immediate execution.

Parameters **selector** – An optional single argument function which will be used to project the elements of the sequence. If omitted, the identity function is used.
Returns The arithmetic mean value of the projected sequence.
Raises
- ValueError - If the Queryable has been closed.
- ValueError - I the source sequence is empty.

Examples

Compute the average of some numbers:

```
>>> numbers = [98, 458, 32, 876, 12, 9, 325]
>>> query(numbers).average()
258.57142857142856
```

Compute the mean square of a sequence:

```
>>> numbers = [98, 458, 32, 876, 12, 9, 325]
>>> query(numbers).average(lambda x: x*x)
156231.14285714287
```

close()
Closes the queryable.

The Queryable should not be used following a call to close. This method is idempotent. Other calls to a Queryable following close() will raise ValueError.

closed()
Determine whether the Queryable has been closed.
Returns True if closed, otherwise False.

concat(*second_iterable*)
Concatenates two sequences.

Note: This method uses deferred execution.

Parameters **second_iterable** – The sequence to concatenate on to the sequence.
Returns A Queryable over the concatenated sequences.
Raises
- ValueError - If the Queryable is closed().
- TypeError - If second_iterable is not in fact iterable.

Example

Concatenate two sequences of numbers:

```
>>> numbers = [1, 45, 23, 34]
>>> query(numbers).concat([98, 23, 23, 12]).to_list()
[1, 45, 23, 34, 98, 23, 23, 12]
```

contains (*value*, *equality_comparer=operator.eq*)
Determines whether the sequence contains a particular value.

Execution is immediate. Depending on the type of the sequence, all or none of the sequence may be consumed by this operation.

Note: This method uses immediate execution.

Parameters **value** – The value to test for membership of the sequence
Returns True if value is in the sequence, otherwise False.
Raises ValueError - If the Queryable has been closed.

Example

Check whether a sentence contains a particular word:

```
>>> words = ['A', 'man', 'a', 'plan', 'a', 'canal', 'Panama']
>>> words.contains('plan')
True
```

Check whether a sentence contains a particular word with a case- insensitive check:

```
>>> words = ['A', 'man', 'a', 'plan', 'a', 'canal', 'Panama']
>>> query(words).contains('panama',
...                       lambda lhs, rhs: lhs.lower() == rhs.lower())
True
```

count (*predicate=None*)
Return the number of elements (which match an optional predicate).

Note: This method uses immediate execution.

Parameters **predicate** – An optional unary predicate function used to identify elements which will be counted. The single positional argument of the function is the element value. The function should return True or False.

Returns The number of elements in the sequence if the predicate is None (the default), or if the predicate is supplied the number of elements for which the predicate evaluates to True.

Raises

- ValueError - If the Queryable is closed().
- TypeError - If predicate is neither None nor a callable.

Examples

Count the number of elements in a sequence:

```
>>> people = ['Sheila', 'Jim', 'Fred']
>>> query(people).count()
3
```

Count the number of names containing the letter 'i':

```
>>> people = ['Sheila', 'Jim', 'Fred']
>>> query(people).count(lambda s: 'i' in s)
3
```

default_if_empty(*default*)

If the source sequence is empty return a single element sequence containing the supplied default value, otherwise return the source sequence unchanged.

Note: This method uses deferred execution.

Parameters **default** – The element to be returned if the source sequence is empty.

Returns The source sequence, or if the source sequence is empty an sequence containing a single element with the supplied default value.

Raises ValueError - If the Queryable has been closed.

Examples

An empty sequence triggering the default return:

```
>>> e = []
>>> query(e).default_if_empty(97).to_list()
[97]
```

A non-empty sequence passing through:

```
>>> f = [70, 45, 34]
>>> query(f).default_if_empty(97).to_list()
[70, 45, 34]
```

difference(*second_iterable*, *selector=identity*)

Returns those elements which are in the source sequence which are not in the second_iterable.

This method is equivalent to the Except() LINQ operator, renamed to a valid Python identifier.

Note: This method uses deferred execution, but as soon as execution commences the entirety of the second_iterable is consumed; therefore, although the source sequence may be infinite the

second_iterable must be finite.

Parameters
- **second_iterable** – Elements from this sequence are excluded from the returned sequence. This sequence will be consumed in its entirety, so must be finite.
- **selector** – A optional single argument function with selects from the elements of both sequences the values which will be compared for equality. If omitted the identity function will be used.

Returns A sequence containing all elements in the source sequence except those which are also members of the second sequence.

Raises
- ValueError - If the Queryable has been closed.
- TypeError - If the second_iterable is not in fact iterable.
- TypeError - If the selector is not callable.

Examples

Numbers in the first list which are not in the second list:

```
>>> a = [0, 2, 4, 5, 6, 8, 9]
>>> b = [1, 3, 5, 7, 8]
>>> query(a).difference(b).to_list()
[0, 2, 4, 6, 9]
```

Countries in the first list which are not in the second list, compared in a case-insensitive manner:

```
>>> a = ['UK', 'Canada', 'qatar', 'china', 'New Zealand', 'Iceland']
>>> b = ['iceland', 'CANADA', 'uk']
>>> query(a).difference(b, lambda x: x.lower()).to_list()
['qatar', 'china', 'New Zealand']
```

distinct(*selector=identity*)

Eliminate duplicate elements from a sequence.

Note: This method uses deferred execution.

Parameters **selector** – An optional single argument function the result of which is the value compared for uniqueness against elements already consumed. If omitted, the element value itself is compared for uniqueness.

Returns Unique elements of the source sequence as determined by the selector function. Note that it is unprojected elements that are returned, even if a selector was provided.

Raises
- ValueError - If the Queryable is closed.
- TypeError - If the selector is not callable.

Examples

Remove duplicate numbers:

```
>>> d = [0, 2, 4, 5, 6, 8, 9, 1, 3, 5, 7, 8]
>>> query(d).distinct().to_list()
[0, 2, 4, 5, 6, 8, 9, 1, 3, 7]
```

A sequence such that no two numbers in the result have digits which sum to the same value:

```
>>> e = [10, 34, 56, 43, 74, 25, 11, 89]
>>> def sum_of_digits(num):
...     return sum(int(i) for i in str(num))
...
>>> query(e).distinct(sum_of_digits).to_list()
[10, 34, 56, 11, 89]
```

element_at (*index*)

Return the element at ordinal index.

Note: This method uses immediate execution.

Parameters **index** – The index of the element to be returned.
Returns The element at ordinal index in the source sequence.
Raises

- ValueError - If the Queryable is closed().
- ValueError - If index is out of range.

Example

Retrieve the fifth element from a list:

```
>>> f = [10, 34, 56, 11, 89]
>>> query(f).element_at(4)
89
```

first (*predicate=None*)

The first element in a sequence (optionally satisfying a predicate).

If the predicate is omitted or is None this query returns the first element in the sequence; otherwise, it returns the first element in the sequence for which the predicate evaluates to True. Exceptions are raised if there is no such element.

Note: This method uses immediate execution.

Parameters **predicate** – An optional unary predicate function, the only argument to which is the element. The return value should be True for matching elements, otherwise False. If the predicate is omitted or None the first element of the source sequence will be returned.
Returns The first element of the sequence if predicate is None, otherwise the first element for which the predicate returns True.
Raises

- ValueError - If the Queryable is closed.
- ValueError - If the source sequence is empty.
- ValueError - If there are no elements matching the predicate.
- TypeError - If the predicate is not callable.

Examples

Retrieve the first element of a sequence:

```
>>> e = [10, 34, 56, 43, 74, 25, 11, 89]
>>> query(e).first()
10
```

Retrieve the first element of a sequence divisible by seven:

```
>>> e = [10, 34, 56, 43, 74, 25, 11, 89]
>>> query(e).first(lambda x: x % 7 == 0)
56
```

first_or_default(*default*, *predicate=None*)

The first element (optionally satisfying a predicate) or a default.

If the predicate is omitted or is None this query returns the first element in the sequence; otherwise, it returns the first element in the sequence for which the predicate evaluates to True. If there is no such element the value of the default argument is returned.

Note: This method uses immediate execution.

Parameters

- **default** – The value which will be returned if either the sequence is empty or there are no elements matching the predicate.
- **predicate** – An optional unary predicate function, the only argument to which is the element. The return value should be True for matching elements, otherwise False. If the predicate is omitted or None the first element of the source sequence will be returned.

Returns The first element of the sequence if predicate is None, otherwise the first element for which the predicate returns True. If there is no such element, the default argument is returned.

Raises

- ValueError - If the Queryable is closed.
- TypeError - If the predicate is not callable.

Examples

Retrieve the first element of a sequence:

```
>>> e = [10, 34, 56, 43, 74, 25, 11, 89]
>>> query(e).first_or_default(14)
10
```

Return the default when called on an empty sequence:

```
>>> f = []
>>> query(f).first_or_default(17)
17
```

Retrieve the first element of a sequence divisible by eight:

```
>>> e = [10, 34, 56, 43, 74, 25, 11, 89]
>>> query(e).first_or_default(10, lambda x: x % 8 == 0)
56
```

group_by(*key_selector=identity*, *element_selector=identity*, *result_selector=lambda key, grouping: grouping*)

Groups the elements according to the value of a key extracted by a selector function.

Note: This method has different behaviour to itertools.groupby in the Python standard library because it aggregates all items with the same key, rather than returning groups of consecutive items of the same key.

Note: This method uses deferred execution, but consumption of a single result will lead to evaluation of the whole source sequence.

Parameters

- **key_selector** – An optional unary function used to extract a key from each element in the source sequence. The default is the identity function.
- **element_selector** – A optional unary function to map elements in the source sequence to elements in a resulting Grouping. The default is the identity function.
- **result_selector** – An optional binary function to create a result from each group. The first positional argument is the key identifying the group. The second argument is a Grouping object containing the members of the group. The default is a function which simply returns the Grouping.

Returns A Queryable sequence of elements of the where each element represents a group. If the default result_selector is relied upon this is a Grouping object.

Raises

- ValueError - If the Queryable is closed().
- TypeError - If key_selector is not callable.
- TypeError - If element_selector is not callable.
- TypeError - If result_selector is not callable.

Examples

Group numbers by the remainder when dividing them by five:

```
>>> numbers = [10, 34, 56, 43, 74, 25, 11, 89]
>>> groups = query(e).group_by(lambda x: x % 5).to_list()
>>> groups
[Grouping(key=0), Grouping(key=4), Grouping(key=1),
 Grouping(key=3)]
>>> groups[0].key
0
>>> groups[0].to_list()
[10, 25]
>>> groups[1].key
1
>>> groups[1].to_list()
[34, 74, 89]
```

Group people by their nationality of the first name, and place only the person's name in the grouped result:

```
>>> people = [ dict(name="Joe Bloggs", nationality="British"),
...            dict(name="Ola Nordmann", nationality="Norwegian"),
...            dict(name="Harry Holland", nationality="Dutch"),
...            dict(name="Kari Nordmann", nationality="Norwegian"),
...            dict(name="Jan Kowalski", nationality="Polish"),
...            dict(name="Hans Schweizer", nationality="Swiss"),
...            dict(name="Tom Cobbleigh", nationality="British"),
...            dict(name="Tommy Atkins", nationality="British") ]
>>> groups = query(people).group_by(lambda p: p['nationality'],
                                    lambda p: p['name']).to_list()
>>> groups
[Grouping(key='British'), Grouping(key='Norwegian'),
 Grouping(key='Dutch'), Grouping(key='Polish'),
 Grouping(key='Swiss')]
>>> groups[0].to_list()
['Joe Bloggs', 'Tom Cobbleigh', 'Tommy Atkins']
>>> groups[1].to_list()
['Ola Nordmann', 'Kari Nordmann']
```

Determine the number of people in each national group by creating a tuple for each group where the first element is the nationality and the second element is the number of people of that nationality:

```
>>> people = [ dict(name="Joe Bloggs", nationality="British"),
...            dict(name="Ola Nordmann", nationality="Norwegian"),
...            dict(name="Harry Holland", nationality="Dutch"),
...            dict(name="Kari Nordmann", nationality="Norwegian"),
...            dict(name="Jan Kowalski", nationality="Polish"),
...            dict(name="Hans Schweizer", nationality="Swiss"),
...            dict(name="Tom Cobbleigh", nationality="British"),
...            dict(name="Tommy Atkins", nationality="British") ]
>>> groups = query(people).group_by(lambda p: p['nationality'],
...  result_selector=lambda key, group: (key, len(group))).to_list()
>>> groups
[('British', 3), ('Norwegian', 2), ('Dutch', 1), ('Polish', 1),
 ('Swiss', 1)]
```

group_join(*inner_iterable*, *outer_key_selector=identity*, *inner_key_selector=identity*, *result_selector=lambda outer*, *grouping: grouping*)

Match elements of two sequences using keys and group the results.

The group_join() query produces a hierarchical result, with all of the inner elements in the result grouped against the matching outer element.

The order of elements from outer is maintained. For each of these the order of elements from inner is also preserved.

Note: This method uses deferred execution.

Parameters

- **inner_iterable** – The sequence to join with the outer sequence.
- **outer_key_selector** – An optional unary function to extract keys from elements of the outer (source) sequence. The first positional argument of the function should accept outer elements and the result value should be the key. If omitted, the identity function is used.
- **inner_key_selector** – An optional unary function to extract keys from el-

ements of the inner_iterable. The first positional argument of the function should accept outer elements and the result value should be the key. If omitted, the identity function is used.

- **result_selector** – An optional binary function to create a result element from an outer element and the Grouping of matching inner elements. The first positional argument is the outer elements and the second in the Grouping of inner elements which match the outer element according to the key selectors used. If omitted, the result elements will be the Groupings directly.

Returns A Queryable over a sequence with one element for each group in the result as returned by the result_selector. If the default result selector is used, the result is a sequence of Grouping objects.

Raises

- ValueError - If the Queryable has been closed.
- TypeError - If the inner_iterable is not in fact iterable.
- TypeError - If the outer_key_selector is not callable.
- TypeError - If the inner_key_selector is not callable.
- TypeError - If the result_selector is not callable.

Example

Correlate players with soccer teams using the team name. Group the players within those teams such that each element of the result sequence contains full information about a team and a collection of players belonging to that team:

```
>>> players = [dict(name="Ferdinand", team="Manchester United"),
...            dict(name="Cole", team="Chelsea", fee=5),
...            dict(name="Crouch", team="Tottenham Hotspur"),
...            dict(name="Downing", team="Aston Villa"),
...            dict(name="Lampard", team="Chelsea", fee=11),
...            dict(name="Rooney", team="Manchester United"),
...            dict(name="Scholes", team="Manchester United",␣
↪fee=None)]
>>> teams = [dict(name="Manchester United", ground="Old Trafford"),
...          dict(name="Chelsea", ground="Stamford Bridge"),
...          dict(name="Tottenham Hotspur", ground="White Hart Lane"),
...          dict(name="Aston Villa", ground="Villa Park")]
>>> q = query(teams).group_join(players, lambda team: team['name'],
...                lambda player: player['team'],
...                lambda team, grouping: dict(team=team['name'],
...                                            ground=team['ground'],
...                                            players=grouping)).to_
↪list()
>>> q
[{'players': Grouping(key='Manchester United'), 'ground': 'Old␣
↪Trafford', 'team': 'Manchester United'},
 {'players': Grouping(key='Chelsea'), 'ground': 'Stamford Bridge',
↪'team': 'Chelsea'},
 {'players': Grouping(key='Tottenham Hotspur'), 'ground': 'White Hart␣
↪Lane', 'team': 'Tottenham Hotspur'},
 {'players': Grouping(key='Aston Villa'), 'ground': 'Villa Park',
↪'team': 'Aston Villa'}]
>>> q[0]['players'].to_list()
[{'name': 'Ferdinand', 'team': 'Manchester United'},
 {'name': 'Rooney', 'team': 'Manchester United'},
 {'name': 'Scholes', 'team': 'Manchester United'}]
```

intersect(*second_iterable*, *selector=identity*)
Returns those elements which are both in the source sequence and in the second_iterable.

Note: This method uses deferred execution.

Parameters
- **second_iterable** – Elements are returned if they are also in the sequence.
- **selector** – An optional single argument function which is used to project the elements in the source and second_iterables prior to comparing them. If omitted the identity function will be used.

Returns A sequence containing all elements in the source sequence which are also members of the second sequence.

Raises
- ValueError - If the Queryable has been closed.
- TypeError - If the second_iterable is not in fact iterable.
- TypeError - If the selector is not callable.

Examples

Find all the numbers common to both lists `a` and `b`:

```
>>> a = [1, 6, 4, 2, 6, 7, 3, 1]
>>> b = [6, 2, 1, 9, 2, 5]
>>> query(a).intersect(b).to_list()
[1, 6, 2]
```

Take those strings from the list `a` which also occur in list `b` when compared in a case-insensitive way:

```
>>> a = ["Apple", "Pear", "Banana", "Orange", "Strawberry"]
>>> b = ["PEAR", "ORANGE", "BANANA", "RASPBERRY", "BLUEBERRY"]
>>> query(a).intersect(b, lambda s: s.lower()).to_list()
['Pear', 'Banana', 'Orange']
```

join(*inner_iterable*, *outer_key_selector=identity*, *inner_key_selector=identity*, *result_selector=lambda outer*, *inner: (outer*, *inner)*)
Perform an inner join with a second sequence using selected keys.

The order of elements from outer is maintained. For each of these the order of elements from inner is also preserved.

Note: This method uses deferred execution.

Parameters
- **inner_iterable** – The sequence to join with the outer sequence.
- **outer_key_selector** – An optional unary function to extract keys from elements of the outer (source) sequence. The first positional argument of the function should accept outer elements and the result value should be the key. If omitted, the identity function is used.
- **inner_key_selector** – An optional unary function to extract keys from elements of the inner_iterable. The first positional argument of the function should accept outer elements and the result value should be the key. If omitted, the identity function is used.

- **result_selector** – An optional binary function to create a result element from two matching elements of the outer and inner. If omitted the result elements will be a 2-tuple pair of the matching outer and inner elements.

Returns A Queryable whose elements are the result of performing an inner- join on two sequences.

Raises

- ValueError - If the Queryable has been closed.
- TypeError - If the inner_iterable is not in fact iterable.
- TypeError - If the outer_key_selector is not callable.
- TypeError - If the inner_key_selector is not callable.
- TypeError - If the result_selector is not callable.

Examples

Correlate pets with their owners, producing pairs of pet and owner date for each result:

```
>>> people = ['Minnie', 'Dennis', 'Roger', 'Beryl']
>>> pets = [dict(name='Chester', owner='Minnie'),
...         dict(name='Gnasher', owner='Dennis'),
...         dict(name='Dodge', owner='Roger'),
...         dict(name='Pearl', owner='Beryl')]
>>> query(pets).join(people, lambda pet: pet['owner']).to_list()
[({'owner': 'Minnie', 'name': 'Chester'}, 'Minnie'),
 ({'owner': 'Dennis', 'name': 'Gnasher'}, 'Dennis'),
 ({'owner': 'Roger', 'name': 'Dodge'}, 'Roger'),
 ({'owner': 'Beryl', 'name': 'Pearl'}, 'Beryl')]
```

or correlate owners with pets, producing more refined results:

```
>>> query(people).join(pets, inner_key_selector=lambda pet: pet['owner
↪'],
...   result_selector=lambda person, pet: pet['name'] + " is owned by
↪" + person) \
...   .to_list()
['Chester is owned by Minnie',
 'Gnasher is owned by Dennis',
 'Dodge is owned by Roger',
 'Pearl is owned by Beryl']
```

last (*predicate=None*)

The last element in a sequence (optionally satisfying a predicate).

If the predicate is omitted or is None this query returns the last element in the sequence; otherwise, it returns the last element in the sequence for which the predicate evaluates to True. Exceptions are raised if there is no such element.

Note: This method uses immediate execution.

Parameters **predicate** – An optional unary predicate function, the only argument to which is the element. The return value should be True for matching elements, otherwise False. If the predicate is omitted or None the last element of the source sequence will be returned.

Returns The last element of the sequence if predicate is None, otherwise the last element for which the predicate returns True.

Raises

- ValueError - If the Queryable is closed.
- ValueError - If the source sequence is empty.
- ValueError - If there are no elements matching the predicate.
- TypeError - If the predicate is not callable.

Examples

Return the last number in this sequence:

```
>>> numbers = [1, 45, 23, 34]
>>> query(numbers).last()
34
```

Return the last number under 30 in this sequence:

```
>>> numbers = [1, 45, 23, 34]
>>> query(numbers).last(lambda x: x < 30)
23
```

last_or_default(*default*, *predicate=None*)

The last element (optionally satisfying a predicate) or a default.

If the predicate is omitted or is None this query returns the last element in the sequence; otherwise, it returns the last element in the sequence for which the predicate evaluates to True. If there is no such element the value of the default argument is returned.

Note: This method uses immediate execution.

Parameters

- **default** – The value which will be returned if either the sequence is empty or there are no elements matching the predicate.
- **predicate** – An optional unary predicate function, the only argument to which is the element. The return value should be True for matching elements, otherwise False. If the predicate is omitted or None the last element of the source sequence will be returned.

Returns The last element of the sequence if predicate is None, otherwise the last element for which the predicate returns True. If there is no such element, the default argument is returned.

Raises

- ValueError - If the Queryable is closed.
- TypeError - If the predicate is not callable.

Examples

Return the last number in this sequence:

```
>>> numbers = [1, 45, 23, 34]
>>> query(numbers).last()
34
```

Return the last number under 30 in this sequence:

```
>>> numbers = [1, 45, 23, 34]
>>> query(numbers).last(lambda x: x < 30)
23
```

Trigger return of the default using a sequence with no values which satisfy the predicate:

```
>>> numbers = [1, 45, 23, 34]
>>> query(numbers).last_or_default(100, lambda x: x > 50)
100
```

Trigger return of the default using an empty sequence:

```
>>> numbers = []
>>> query(numbers).last_or_default(37)
37
```

log(*logger=None*, *label=None*, *eager=False*)

Log query result consumption details to a logger.

Parameters

- **logger** – Any object which supports a debug() method which accepts a str, such as a Python standard library logger object from the logging module. If logger is not provided or is None, this method has no logging side effects.
- **label** – An optional label which will be inserted into each line of logging output produced by this particular use of log
- **eager** – An optional boolean which controls how the query result will be consumed. If True, the sequence will be consumed and logged in its entirety. If False (the default) the sequence will be evaluated and logged lazily as it consumed.

Warning: Use of eager=True requires use of sufficient memory to hold the entire sequence which is obviously not possible with infinite sequences. Use with care!

Returns A queryable over the unaltered source sequence.

Raises

- AttributeError - If logger does not support a debug() method.
- ValueError - If the Queryable has been closed.

Examples

These examples log to a console logger called `clog` which can be created using the following incantation:

```
>>> import logging
>>> clog = logging.getLogger("clog")
>>> clog.setLevel(logging.DEBUG)
>>> clog.addHandler(logging.StreamHandler())
```

By default, `log()` uses deferred execution, so unless the output of `log()` is consumed nothing at all will be logged. In this example nothing is logged to the console because the result of `log()` is never consumed:

```
>>> numbers = [1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]
>>> query(numbers).log(clog)
```

We can easily consume the output of `log()` by chaining a call to `to_list()`. Use the default arguments for `log()`:

```
>>> numbers = [1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]
>>> query(numbers).log(clog).to_list()
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪BEGIN (DEFERRED)
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[0] yields 1
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[1] yields 5
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[2] yields 9
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[3] yields 34
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[4] yields 2
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[5] yields 9
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[6] yields 12
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[7] yields 7
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[8] yields 13
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[9] yields 48
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[10] yields 34
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[11] yields 23
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[12] yields 34
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[13] yields 9
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪[14] yields 47
Queryable([1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]) :␣
↪END (DEFERRED)
[1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]
```

The beginning and end of the sequence are delimited by `BEGIN` and `END` markers which also indicated whether logging is `DEFERRED` so items are logged only as they are requested or `EAGER` where the whole sequence will be returns immediately.

From left to right the log output shows:

1. A label, which defaults to the `repr()` of the Queryable instance being logged.
2. In square brackets the zero-based index of the element being logged.
3. `yields <element>` showing the element value

Specify a label a more concise label to `log()`:

```
>>> numbers = [1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]
>>> query(numbers).log(clog, label='query()').to_list()
query() : BEGIN (DEFERRED)
query() : [0] yields 1
query() : [1] yields 5
query() : [2] yields 9
query() : [3] yields 34
```

```
query() : [4] yields 2
query() : [5] yields 9
query() : [6] yields 12
query() : [7] yields 7
query() : [8] yields 13
query() : [9] yields 48
query() : [10] yields 34
query() : [11] yields 23
query() : [12] yields 34
query() : [13] yields 9
query() : [14] yields 47
query() : END (DEFERRED)
[1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]
```

We can show how the default deferred logging produces only required elements by only consuming the first three elements:

```
>>> numbers = [1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]
>>> query(numbers).log(clog, label='query()').take(3).to_list()
query() : BEGIN (DEFERRED)
query() : [0] yields 1
query() : [1] yields 5
query() : [2] yields 9
[1, 5, 9]
```

However, by setting the `eager` argument to be True, we can force logging of the whole sequence immediately:

```
>>> numbers = [1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]
>>> query(numbers).log(clog, label='query()', eager=True).take(3).to_
↪list()
query() : BEGIN (EAGER)
query() : [0] = 1
query() : [1] = 5
query() : [2] = 9
query() : [3] = 34
query() : [4] = 2
query() : [5] = 9
query() : [6] = 12
query() : [7] = 7
query() : [8] = 13
query() : [9] = 48
query() : [10] = 34
query() : [11] = 23
query() : [12] = 34
query() : [13] = 9
query() : [14] = 47
query() : END (EAGER)
[1, 5, 9]
```

Note that in these cases the output has a different format and that use of eager logging in no way affects the query result.

If `logger` is None (or omitted), then logging is disabled completely:

```
>>> query(numbers).log(logger=None, label='query()').take(3).to_list()
[1, 5, 9]
```

Finally, see that `log()` can be used at multiple points within a query expression:

```
>>> numbers = [1, 5, 9, 34, 2, 9, 12, 7, 13, 48, 34, 23, 34, 9, 47]
>>> query(numbers).log(clog, label='query(numbers)')
↪ \
...         .select(lambda x: x * x).log(clog, label='squared')     \
...         .where(lambda x: x > 1000).log(clog, label="over 1000") \
...         .take(3).log(clog, label="take 3")                      \
...         .to_list()
take 3 : BEGIN (DEFERRED)
over 1000 : BEGIN (DEFERRED)
squared : BEGIN (DEFERRED)
query(numbers) : BEGIN (DEFERRED)
query(numbers) : [0] yields 1
squared : [0] yields 1
query(numbers) : [1] yields 5
squared : [1] yields 25
query(numbers) : [2] yields 9
squared : [2] yields 81
query(numbers) : [3] yields 34
squared : [3] yields 1156
over 1000 : [0] yields 1156
take 3 : [0] yields 1156
query(numbers) : [4] yields 2
squared : [4] yields 4
query(numbers) : [5] yields 9
squared : [5] yields 81
query(numbers) : [6] yields 12
squared : [6] yields 144
query(numbers) : [7] yields 7
squared : [7] yields 49
query(numbers) : [8] yields 13
squared : [8] yields 169
query(numbers) : [9] yields 48
squared : [9] yields 2304
over 1000 : [1] yields 2304
take 3 : [1] yields 2304
query(numbers) : [10] yields 34
squared : [10] yields 1156
over 1000 : [2] yields 1156
take 3 : [2] yields 1156
take 3 : END (DEFERRED)
[1156, 2304, 1156]
```

max(*selector=identity*)

Return the maximum value in a sequence.

All of the source sequence will be consumed.

Note: This method uses immediate execution.

Parameters **selector** – An optional single argument function which will be used to project the elements of the sequence. If omitted, the identity function is used.

Returns The maximum value of the projected sequence.

Raises

- ValueError - If the Queryable has been closed.
- ValueError - If the sequence is empty.

Examples

Return the maximum value from a list of numbers:

```
>>> numbers = [1, -45, 23, -34, 19]
>>> query(numbers).max()
23
```

Return the maximum absolute value from a list of numbers:

```
>>> numbers = [1, -45, 23, -34, 19]
>>> query(numbers).max(abs)
45
```

min (*selector=identity*)

Return the minimum value in a sequence.

All of the source sequence will be consumed.

Note: This method uses immediate execution.

Parameters **selector** – An optional single argument function which will be used to project the elements of the sequence. If omitted, the identity function is used.

Returns The minimum value of the projected sequence.

Raises

- ValueError - If the Queryable has been closed.
- ValueError - If the sequence is empty.

Examples

Return the minimum value from a list of numbers:

```
>>> numbers = [1, -45, 23, -34, 19]
>>> query(numbers).max()
-45
```

Return the minimum absolute value from a list of numbers:

```
>>> numbers = [1, -45, 23, -34, 19]
>>> query(numbers).max(abs)
1
```

of_type (*classinfo*)

Filters elements according to whether they are of a certain type.

Note: This method uses deferred execution.

Parameters **classinfo** – If classinfo is neither a class object nor a type object it may be a tuple of class or type objects, or may recursively contain other such tuples (other sequence types are not accepted).

Returns A Queryable over those elements of the source sequence for which the predicate is True.

Raises

- ValueError - If the Queryable is closed.
- **TypeError - If classinfo is not a class, type, or tuple of classes,** types, and such tuples.

Examples

Return all of the strings from a list:

```
>>> numbers = ["one", 2.0, "three", "four", 5, 6.0, "seven", 8, "nine
↪", "ten"]
>>> query(numbers).of_type(int).to_list()
[5, 8]
```

Return all the integers and floats from a list:

```
>>> numbers = ["one", 2.0, "three", "four", 5, 6.0, "seven", 8, "nine
↪", "ten"]
>>> query(numbers).of_type((int, float)).to_list()
[2.0, 5, 6.0, 8]
```

order_by (*key_selector=identity*)

Sorts by a key in ascending order.

Introduces a primary sorting order to the sequence. Additional sort criteria should be specified by subsequent calls to then_by() and then_by_descending(). Calling order_by() or order_by_descending() on the results of a call to order_by() will introduce a new primary ordering which will override any already established ordering.

This method performs a stable sort. The order of two elements with the same key will be preserved.

Note: This method uses deferred execution.

Parameters **key_selector** – A unary function which extracts a key from each element using which the result will be ordered.

Returns An OrderedQueryable over the sorted elements.

Raises

- ValueError - If the Queryable is closed.
- TypeError - If the key_selector is not callable.

Examples

Sort a list of numbers in ascending order by their own value:

```
>>> numbers = [1, -45, 23, -34, 19, 78, -23, 12, 98, -14]
>>> query(numbers).order_by().to_list()
[-45, -34, -23, -14, 1, 12, 19, 23, 78, 98]
```

Sort a list of numbers in ascending order by their absolute value:

```
>>> numbers = [1, -45, 23, -34, 19, 78, -23, 12, 98, -14]
>>> query(numbers).order_by(abs).to_list()
[1, 12, -14, 19, 23, -23, -34, -45, 78, 98]
```

See that the relative order of the two elements which compare equal (23 and -23 in the list shown) are preserved; the sort is stable.

order_by_descending(*key_selector=identity*)

Sorts by a key in descending order.

Introduces a primary sorting order to the sequence. Additional sort criteria should be specified by subsequent calls to then_by() and then_by_descending(). Calling order_by() or order_by_descending() on the results of a call to order_by() will introduce a new primary ordering which will override any already established ordering.

This method performs a stable sort. The order of two elements with the same key will be preserved.

Note: This method uses deferred execution.

Parameters **key_selector** – A unary function which extracts a key from each element using which the result will be ordered.

Returns An OrderedQueryable over the sorted elements.

Raises

- ValueError - If the Queryable is closed.
- TypeError - If the key_selector is not callable.

Examples

Sort a list of numbers in ascending order by their own value:

```
>>> numbers = [1, -45, 23, -34, 19, 78, -23, 12, 98, -14]
>>> query(numbers).order_by_descending().to_list()
[98, 78, 23, 19, 12, 1, -14, -23, -34, -45]
```

Sort a list of numbers in ascending order by their absolute value:

```
>>> numbers = [1, -45, 23, -34, 19, 78, -23, 12, 98, -14]
>>> query(numbers).order_by_descending(abs).to_list()
[98, 78, -45, -34, 23, -23, 19, -14, 12, 1]
```

See that the relative order of the two elements which compare equal (23 and -23 in the list shown) are preserved; the sort is stable.

select(*selector*)

Transforms each element of a sequence into a new form.

Each element of the source is transformed through a selector function to produce a corresponding element in teh result sequence.

If the selector is identity the method will return self.

Note: This method uses deferred execution.

Parameters **selector** – A unary function mapping a value in the source sequence to the corresponding value in the generated generated sequence. The single positional argument to the selector function is the element value. The return value of the selector function should be the corresponding element of the result sequence.

Returns A Queryable over generated sequence whose elements are the result of invoking the selector function on each element of the source sequence.

Raises

- ValueError - If this Queryable has been closed.
- TypeError - If selector is not callable.

Examples

Select the scores from a collection of student records:

```
>>> students = [dict(name="Joe Bloggs", score=54),
...             dict(name="Ola Nordmann", score=61),
...             dict(name="John Doe", score=51),
...             dict(name="Tom Cobleigh", score=71)]
>>> query(students).select(lambda student: student['score']).to_
↪list()
[54, 61, 51, 71]
```

Transform a sequence of numbers into it square roots:

```
>>> import math
>>> numbers = [1, 45, 23, 34, 19, 78, 23, 12, 98, 14]
>>> query(numbers).select(math.sqrt).to_list()
[1.0, 6.708203932499369, 4.795831523312719, 5.830951894845301,
 4.358898943540674, 8.831760866327848, 4.795831523312719,
 3.4641016151377544, 9.899494936611665, 3.7416573867739413]
```

select_many(*collection_selector=identity*, *result_selector=identity*)

Projects each element of a sequence to an intermediate new sequence, flattens the resulting sequences into one sequence and optionally transforms the flattened sequence using a selector function.

Note: This method uses deferred execution.

Parameters

- **collection_selector** – A unary function mapping each element of the source iterable into an intermediate sequence. The single argument of the collection_selector is the value of an element from the source sequence. The return value should be an iterable derived from that element value. The default collection_selector, which is the identity function, assumes that each element of the source sequence is itself iterable.

- **result_selector** – An optional unary function mapping the elements in the flattened intermediate sequence to corresponding elements of the result sequence. The single argument of the result_selector is the value of an element from the flattened intermediate sequence. The return value should be the corresponding value in the result sequence. The default result_selector is the identity function.

Returns A Queryable over a generated sequence whose elements are the result of applying the one-to-many collection_selector to each element of the source sequence, concatenating the results into an intermediate sequence, and then mapping each of those elements through the result_selector into the result sequence.

Raises

- ValueError - If this Queryable has been closed.
- **TypeError - If either collection_selector or result_selector are not** callable.

Examples

Select all the words from three sentences by splitting each sentence into its component words:

```
>>> a = "The quick brown fox jumped over the lazy dog"
>>> b = "Pack my box with five dozen liquor jugs"
>>> c = "Jackdaws love my big sphinx of quartz"
>>> sentences = [a, b, c]
>>> query(sentences).select_many(lambda sentence: sentence.split()).
↪to_list()
['The', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy',
 'dog', 'Pack', 'my', 'box', 'with', 'five', 'dozen', 'liquor',
 'jugs', 'Jackdaws', 'love', 'my', 'big', 'sphinx', 'of', 'quartz']
```

Select all the words from three sentences and return a list of the length of each word:

```
>>> a = "The quick brown fox jumped over the lazy dog"
>>> b = "Pack my box with five dozen liquor jugs"
>>> c = "Jackdaws love my big sphinx of quartz"
>>> sentences = [a, b, c]
>>> query(sentences).select_many(lambda sentence: sentence.split(),␣
↪len).to_list()
[3, 5, 5, 3, 6, 4, 3, 4, 3, 4, 2, 3, 4, 4, 5, 6, 4, 8, 4, 2, 3, 6,
 2, 6]
```

select_many_with_correspondence(*collection_selector=identity*, *result_selector=lambda source_element, collection_element: (source_element, collection_element)*)

Projects each element of a sequence to an intermediate new sequence, and flattens the resulting sequence, into one sequence and uses a selector function to incorporate the corresponding source for each item in the result sequence.

Note: This method uses deferred execution.

Parameters

- **collection_selector** – A unary function mapping each element of the source iterable into an intermediate sequence. The single argument of the collection_selector is the value of an element from the source sequence. The return value should be an iterable derived from that element value. The default collection_selector, which is the identity function, assumes that each element of the source sequence is itself iterable.
- **result_selector** – An optional binary function mapping the elements in the flattened intermediate sequence together with their corresponding source elements to elements of the result sequence. The two positional arguments of the result_selector are, first the source element corresponding to an element from the intermediate sequence, and second the actual element from the intermediate sequence. The return value should be the corresponding value in the result sequence. If no result_selector function is provided, the elements of the result sequence are KeyedElement namedtuples.

Returns A Queryable over a generated sequence whose elements are the result of applying the one-to-many collection_selector to each element of the source sequence, concatenating the results into an intermediate sequence, and then mapping each of those elements through the result_selector which incorporates the corresponding source element into the result sequence.

Raises

- ValueError - If this Queryable has been closed.
- TypeError - If projector or selector are not callable.

Example

Incorporate each album track with its performing artist into a descriptive string:

```
>>> albums = [dict(name="Hotel California", artist="The Eagles",
...                tracks=["Hotel California",
...                        "New Kid in Town",
...                        "Life in the Fast Lane",
...                        "Wasted Time"]),
...           dict(name="Revolver", artist="The Beatles",
...                tracks=["Taxman",
...                        "Eleanor Rigby",
...                        "Yellow Submarine",
...                        "Doctor Robert"]),
...           dict(name="Thriller", artist="Michael Jackson",
...                tracks=["Thriller",
...                        "Beat It",
...                        "Billie Jean",
...                        "The Girl Is Mine"])]
>>> query(albums).select_many_with_correspondence(lambda album:␣
↪album['tracks'],
...   lambda album, track: track + " by " + album['artist']).to_
↪list()
['Hotel California by The Eagles', 'New Kid in Town by The Eagles',
 'Life in the Fast Lane by The Eagles', 'Wasted Time by The Eagles',
 'Taxman by The Beatles', 'Eleanor Rigby by The Beatles',
 'Yellow Submarine by The Beatles', 'Doctor Robert by The Beatles',
 'Thriller by Michael Jackson', 'Beat It by Michael Jackson',
 'Billie Jean by Michael Jackson',
 'The Girl Is Mine by Michael Jackson']
```

select_many_with_index(*collection_selector=IndexedElement*, *result_selector=lambda source_element*, *collection_element: collection_element*)

Projects each element of a sequence to an intermediate new sequence, incorporating the index of the element, flattens the resulting sequence into one sequence and optionally transforms the flattened sequence using a selector function.

Note: This method uses deferred execution.

Parameters

- **collection_selector** – A binary function mapping each element of the source sequence into an intermediate sequence, by incorporating its index in the source sequence. The two positional arguments to the function are the zero-based index of the source element and the value of the element. The result of the function should be an iterable derived from the index and element value. If no collection_selector is provided, the elements of the intermediate sequence will consist of tuples of (index, element) from the source sequence.
- **result_selector** – An optional binary function mapping the elements in the flattened intermediate sequence together with their corresponding source elements to elements of the result sequence. The two positional arguments of the result_selector are, first the source element corresponding to an element from the intermediate sequence, and second the actual element from the intermediate sequence. The return value should be the corresponding value in the result sequence. If no result_selector function is provided, the elements of the flattened intermediate sequence are returned untransformed.

Returns A Queryable over a generated sequence whose elements are the result of applying the one-to-many collection_selector to each element of the source sequence which incorporates both the index and value of the source element, concatenating the results into an intermediate sequence, and then mapping each of those elements through the result_selector into the result sequence.

Raises

- ValueError - If this Queryable has been closed.
- TypeError - If projector [and selector] are not callable.

Example

Incorporate the index of each album along with the track and artist for a digital jukebox. A generator expression is used to combine the index with the track name when generating the intermediate sequences from each album which will be concatenated into the final result:

```
>>> albums = [dict(name="Hotel California", artist="The Eagles",
...                tracks=["Hotel California",
...                        "New Kid in Town",
...                        "Life in the Fast Lane",
...                        "Wasted Time"]),
...           dict(name="Revolver", artist="The Beatles",
...                tracks=["Taxman",
...                        "Eleanor Rigby",
```

```
...                          "Yellow Submarine",
...                          "Doctor Robert"]),
...              dict(name="Thriller", artist="Michael Jackson",
...                   tracks=["Thriller",
...                           "Beat It",
...                           "Billie Jean",
...                           "The Girl Is Mine"])]
>>> query(albums).select_many_with_index(lambda index, album:␣
↪(str(index) + ' - ' + track for track in album['tracks'])).to_
↪list()
['0 - Hotel California', '0 - New Kid in Town',
 '0 - Life in the Fast Lane', '0 - Wasted Time', '1 - Taxman',
 '1 - Eleanor Rigby', '1 - Yellow Submarine', '1 - Doctor Robert',
 '2 - Thriller', '2 - Beat It', '2 - Billie Jean',
 '2 - The Girl Is Mine']
```

Incorporate the index of each album along with the track and artist for a digital jukebox. A generator expression defining the collection_selector is used to combine the index with the track name when generating the intermediate sequences from each album which will be concatenated into the final result:

```
>>> albums = [dict(name="Hotel California", artist="The Eagles",
...                tracks=["Hotel California",
...                        "New Kid in Town",
...                        "Life in the Fast Lane",
...                        "Wasted Time"]),
...           dict(name="Revolver", artist="The Beatles",
...                tracks=["Taxman",
...                        "Eleanor Rigby",
...                        "Yellow Submarine",
...                        "Doctor Robert"]),
...           dict(name="Thriller", artist="Michael Jackson",
...                tracks=["Thriller",
...                        "Beat It",
...                        "Billie Jean",
...                        "The Girl Is Mine"])]
>>> query(albums).select_many_with_index(collection_selector=lambda␣
↪index, album: (str(index) + ' - ' + track for track in album[
↪'tracks']),
...     result_selector=lambda album, track: album['name'] + ' - ' +␣
↪track).to_list()
['Hotel California - 0 - Hotel California',
 'Hotel California - 0 - New Kid in Town',
 'Hotel California - 0 - Life in the Fast Lane',
 'Hotel California - 0 - Wasted Time', 'Revolver - 1 - Taxman',
 'Revolver - 1 - Eleanor Rigby', 'Revolver - 1 - Yellow Submarine',
 'Revolver - 1 - Doctor Robert', 'Thriller - 2 - Thriller',
 'Thriller - 2 - Beat It', 'Thriller - 2 - Billie Jean',
 'Thriller - 2 - The Girl Is Mine']
```

select_with_correspondence(*transform*, *selector=KeyedElement*)

Apply a callable to each element in an input sequence, generating a new sequence of 2-tuples where the first element is the input value and the second is the transformed input value.

The generated sequence is lazily evaluated.

Note: This method uses deferred execution.

Parameters

- **selector** – A unary function mapping a value in the source sequence to the second argument of the result selector.
- **result_selector** – A binary callable mapping the of a value in the source sequence and the transformed value to the corresponding value in the generated sequence. The two positional arguments of the selector function are the original source element and the transformed value. The return value should be the corresponding value in the result sequence. The default selector produces a KeyedElement containing the index and the element giving this function similar behaviour to the built-in enumerate().

Returns When using the default selector, a Queryable whose elements are KeyedElements where the first element is from the input sequence and the second is the result of invoking the transform function on the first value.

Raises

- ValueError - If this Queryable has been closed.
- TypeError - If transform is not callable.

Examples

Generate a list of `KeyedElement` items using the default selector:

```
>>> query(range(10)).select_with_correspondence(lambda x: x%5).to_
↪list()
[KeyedElement(key=0, value=0),
 KeyedElement(key=1, value=1),
 KeyedElement(key=2, value=2),
 KeyedElement(key=3, value=3),
 KeyedElement(key=4, value=4),
 KeyedElement(key=5, value=0),
 KeyedElement(key=6, value=1),
 KeyedElement(key=7, value=2),
 KeyedElement(key=8, value=3),
 KeyedElement(key=9, value=4)]
```

Square the integers zero to nine, retaining only those elements for which the square is an odd number:

```
 >>> query(range(10))                              \
 ... .select_with_correspondence(lambda x: x*x)  \
 ... .where(lambda y: y.value%2 != 0)            \
 ... .select(lambda y: y.key)                    \
 ... .to_list()
 ...
[1, 3, 5, 7, 9]
```

select_with_index(*selector=IndexedElement*)

Transforms each element of a sequence into a new form, incorporating the index of the element.

Each element is transformed through a selector function which accepts the element value and its zero-based index in the source sequence. The generated sequence is lazily evaluated.

Note: This method uses deferred execution.

Parameters **selector** – A binary function mapping the index of a value in the source sequence and the element value itself to the corresponding value in the generated sequence. The two positional arguments of the selector function are the zero- based index of the current element and the value of the current element. The return value should be the corresponding value in the result sequence. The default selector produces an IndexedElement containing the index and the element giving this function similar behaviour to the built-in enumerate().

Returns A Queryable whose elements are the result of invoking the selector function on each element of the source sequence

Raises

- ValueError - If this Queryable has been closed.
- TypeError - If selector is not callable.

Examples

Generate a list of `IndexedElement` items using the default selector. The contents of an `IndexedElement` can either be accessed using the named attributes, or through the zero (index) and one (element) indexes:

```
>>> dark_side_of_the_moon = [ 'Speak to Me', 'Breathe', 'On the Run',
... 'Time', 'The Great Gig in the Sky', 'Money', 'Us and Them',
... 'Any Colour You Like', 'Brain Damage', 'Eclipse']
>>> query(dark_side_of_the_moon).select_with_index().to_list()
[IndexedElement(index=0, element='Speak to Me'),
 IndexedElement(index=1, element='Breathe'),
 IndexedElement(index=2, element='On the Run'),
 IndexedElement(index=3, element='Time'),
 IndexedElement(index=4, element='The Great Gig in the Sky'),
 IndexedElement(index=5, element='Money'),
 IndexedElement(index=6, element='Us and Them'),
 IndexedElement(index=7, element='Any Colour You Like'),
 IndexedElement(index=8, element='Brain Damage'),
 IndexedElement(index=9, element='Eclipse')]
```

Generate numbered album tracks using a custom selector:

```
>>> query(dark_side_of_the_moon).select_with_index(lambda index,␣
↪track: str(index) + '. ' + track).to_list()
['0. Speak to Me', '1. Breathe', '2. On the Run', '3. Time',
 '4. The Great Gig in the Sky', '5. Money', '6. Us and Them',
 '7. Any Colour You Like', '8. Brain Damage', '9. Eclipse']
```

sequence_equal (*second_iterable*, *equality_comparer=operator.eq*)

Determine whether two sequences are equal by elementwise comparison.

Sequence equality is defined as the two sequences being equal length and corresponding elements being equal as determined by the equality comparer.

Note: This method uses immediate execution.

Parameters

- **second_iterable** – The sequence which will be compared with the source sequence.
- **equality_comparer** – An optional binary predicate function which is used to compare corresponding elements. Should return True if the elements are equal, otherwise False. The default equality comparer is operator.eq which calls __eq__ on elements of the source sequence with the corresponding element of the second sequence as a parameter.

Returns True if the sequences are equal, otherwise False.

Raises

- ValueError - If the Queryable is closed.
- TypeError - If second_iterable is not in fact iterable.
- TypeError - If equality_comparer is not callable.

Examples

Determine whether lists `a` and `b` are equal:

```
>>> a = [1, 3, 6, 2, 8]
>>> b = [3, 6, 2, 1, 8]
>>> query(a).sequence_equal(b)
False
```

Determine whether lists `a` and `b` and equal when absolute values are compared:

```
>>> a = [1, -3, 6, -2, 8]
>>> b = [-1, 3, -6, 2, -8]
>>> query(a).sequence_equal(b, lambda lhs, rhs: abs(lhs) == abs(rhs))
True
```

single(*predicate=None*)

The only element (which satisfies a condition).

If the predicate is omitted or is None this query returns the only element in the sequence; otherwise, it returns the only element in the sequence for which the predicate evaluates to True. Exceptions are raised if there is either no such element or more than one such element.

Note: This method uses immediate execution.

Parameters **predicate** – An optional unary predicate function, the only argument to which is the element. The return value should be True for matching elements, otherwise False. If the predicate is omitted or None the only element of the source sequence will be returned.

Returns The only element of the sequence if predicate is None, otherwise the only element for which the predicate returns True.

Raises

- ValueError - If the Queryable is closed.
- **ValueError - If, when predicate is None the source sequence contains** more than one element.
- **ValueError - If there are no elements matching the predicate or more** then one element matching the predicate.
- TypeError - If the predicate is not callable.

Examples

Return the only element in the sequence:

```
>>> a = [5]
>>> query(a).single()
5
```

Attempt to get the single element from a sequence with multiple elements:

```
>>> a = [7, 5, 4]
>>> query(a).single()
ValueError: Sequence for single() contains multiple elements.
```

Return the only element in a sequence meeting a condition:

```
>>> a = [7, 5, 4]
>>> query(a).single(lambda x: x > 6)
7
```

Attempt to get the single element from a sequence which meets a condition when in fact multiple elements do so:

```
>>> a = [7, 5, 4]
>>> query(a).single(lambda x: x >= 5)
ValueError: Sequence contains more than one value matching single()
predicate.
```

single_or_default (*default*, *predicate=None*)

The only element (which satisfies a condition) or a default.

If the predicate is omitted or is None this query returns the only element in the sequence; otherwise, it returns the only element in the sequence for which the predicate evaluates to True. A default value is returned if there is no such element. An exception is raised if there is more than one such element.

Note: This method uses immediate execution.

Parameters

- **default** – The value which will be returned if either the sequence is empty or there are no elements matching the predicate.
- **predicate** – An optional unary predicate function, the only argument to which is the element. The return value should be True for matching elements,

otherwise False. If the predicate is omitted or None the only element of the source sequence will be returned.

Returns The only element of the sequence if predicate is None, otherwise the only element for which the predicate returns True. If there are no such elements the default value will returned.

Raises

- ValueError - If the Queryable is closed.
- **ValueError - If, when predicate is None the source sequence contains** more than one element.
- **ValueError - If there is more then one element matching the** predicate.
- TypeError - If the predicate is not callable.

Examples

Return the only element in the sequence:

```
>>> a = [5]
>>> query(a).single_or_default(7)
5
```

Attempt to get the single element from a sequence with *multiple* elements:

```
>>> a = [7, 5, 4]
>>> query(a).single_or_default(9)
ValueError: Sequence for single_or_default() contains multiple
elements
```

Attempt to get the single element from a sequence with *no* elements:

```
>>> a = []
>>> query(a).single_or_default(9)
9
```

Return the only element in a sequence meeting a condition:

```
>>> a = [7, 5, 4]
>>> query(a).single_or_default(9, lambda x: x > 6)
7
```

Attempt to get the single element from a sequence which meets a condition when in fact multiple elements do so:

```
>>> a = [7, 5, 4]
>>> query(a).single(lambda x: x >= 5)
ValueError: Sequence contains more than one value matching
single_or_default() predicate.
```

Attempt to get the single element matching a predicate from a sequence which contains no matching elements:

```
>>> a = [7, 5, 4]
>>> query(a).single_or_default(9, lambda x: x > 20)
9
```

skip(*count=1*)
Skip the first count contiguous elements of the source sequence.

If the source sequence contains fewer than count elements returns an empty sequence and does not raise an exception.

Note: This method uses deferred execution.

Parameters **count** – The number of elements to skip from the beginning of the sequence. If omitted defaults to one. If count is less than one the result sequence will be empty.

Returns A Queryable over the elements of source excluding the first count elements.

Raises ValueError - If the Queryable is closed().

Examples

Skip the first element of a sequence:

```
>>> a = [7, 5, 4]
>>> query(a).skip().to_list()
[5, 4]
```

Skip the first two elements of a sequence:

```
>>> a = [7, 5, 4]
>>> query(a).skip(2).to_list()
[4]
```

skip_while(*predicate*)
Omit elements from the start for which a predicate is True.

Note: This method uses deferred execution.

Parameters **predicate** – A single argument predicate function.

Returns A Queryable over the sequence of elements beginning with the first element for which the predicate returns False.

Raises

- ValueError - If the Queryable is closed().
- TypeError - If predicate is not callable.

Example

Skip while elements start with the letter 'a':

```
>>> words = ['aardvark', 'antelope', 'ape', 'baboon', 'cat',
...          'anaconda', 'zebra']
>>> query(words).skip_while(lambda s: s.startswith('a')).to_list()
['baboon', 'cat', 'anaconda', 'zebra']
```

sum(*selector=identity*)

Return the arithmetic sum of the values in the sequence..

All of the source sequence will be consumed.

Note: This method uses immediate execution.

Parameters **selector** – An optional single argument function which will be used to project the elements of the sequence. If omitted, the identity function is used.

Returns The total value of the projected sequence, or zero for an empty sequence.

Raises ValueError - If the Queryable has been closed.

Examples

Compute the sum of a sequence of floats:

```
>>> numbers = [5.6, 3.4, 2.3, 9.3, 1.7, 2.4]
>>> query(numbers).sum()
24.7
```

Compute the sum of the squares of a sequence of integers:

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> query(numbers).sum(lambda x: x*x)
385
```

take(*count=1*)

Returns a specified number of elements from the start of a sequence.

If the source sequence contains fewer elements than requested only the available elements will be returned and no exception will be raised.

Note: This method uses deferred execution.

Parameters **count** – An optional number of elements to take. The default is one.

Returns A Queryable over the first count elements of the source sequence, or the all elements of elements in the source, whichever is fewer.

Raises ValueError - If the Queryable is closed()

Examples

Take one element from the start of a list:

```
>>> a = [9, 7, 3, 4, 2]
>>> query(a).take().to_list()
[9]
```

Take three elements from the start of a list:

```
>>> query(a).take(3).to_list()
[9, 7, 3]
```

take_while(*predicate*)
Returns elements from the start while the predicate is True.

Note: This method uses deferred execution.

Parameters **predicate** – A function returning True or False with which elements will be tested.

Returns A Queryable over the elements from the beginning of the source sequence for which predicate is True.

Raises

- ValueError - If the Queryable is closed()
- TypeError - If the predicate is not callable.

Example

```
>>> words = ['aardvark', 'antelope', 'ape', 'baboon', 'cat',
...          'anaconda', 'zebra']
>>> query(words).take_while(lambda s: s.startswith('a')).to_list()
['aardvark', 'antelope', 'ape']
```

to_dictionary(*key_selector=identity*, *value_selector=identity*)
Build a dictionary from the source sequence.

Parameters

- **key_selector** – A unary callable to extract a key from each item. By default the key is the item itself.
- **value_selector** – A unary callable to extract a value from each item. By default the value is the item itself.

Note: This method uses immediate execution.

Raises

- ValueError - If the Queryable is closed.
- TypeError - If key_selector is not callable.
- TypeError - If value_selector is not callable.

Examples

Convert to a dictionary using the default key and value selectors:

```
>>> animals = ['aardvark', 'baboon', 'cat', 'dot', 'frog', 'giraffe',
...            'horse', 'iguana']
>>> query(animals).to_dictionary()
{'horse': 'horse', 'aardvark': 'aardvark', 'frog': 'frog', 'cat':
 'cat', 'giraffe': 'giraffe', 'baboon': 'baboon', 'dot': 'dot',
 'iguana': 'iguana'}
```

Convert to a dictionary extracting the first letter as a key:

```
>>> animals = ['aardvark', 'baboon', 'cat', 'dot', 'frog', 'giraffe',
...            'horse', 'iguana']
>>> query(animals).to_dictionary(key_selector=lambda x: x[0])
{'a': 'aardvark', 'c': 'cat', 'b': 'baboon', 'd': 'dot', 'g':
 'giraffe', 'f': 'frog', 'i': 'iguana', 'h': 'horse'}
```

Convert to a dictionary extracting the first letter as a key and converting the value to uppercase:

```
>>> query(animals).to_dictionary(key_selector=lambda x: x[0],
...                              value_selector=lambda x: x.upper())
{'a': 'AARDVARK', 'c': 'CAT', 'b': 'BABOON', 'd': 'DOT', 'g':
 'GIRAFFE', 'f': 'FROG', 'i': 'IGUANA', 'h': 'HORSE'}
```

Attempt to convert a list of fruit to a dictionary using the initial letter as the key, in the presence of a multiple keys of the same value:

```
>>> fruit = ['apple', 'apricot', 'banana', 'cherry']
>>> query(fruit).to_dictionary(lambda f: f[0])
ValueError: Duplicate key value 'a' in sequence during
to_dictionary()
```

to_list()

Convert the source sequence to a list.

Note: This method uses immediate execution.

Example

Convert from a tuple into a list:

```
>>> a = (1, 6, 8, 3, 4)
>>> query(a).to_list()
[1, 6, 8, 3, 4]
```

to_lookup()

Returns a Lookup object, using the provided selector to generate a key for each item.

Note: This method uses immediate execution.

Examples

Convert to a Lookup using the default key_selector and value_selector:

```
>>> countries = ['Austria', 'Bahrain', 'Canada', 'Algeria',
...              'Belgium', 'Croatia', 'Kuwait', 'Angola', 'Greece',
...              'Korea']
>>> query(countries).to_lookup()
Lookup([('Austria', 'Austria'), ('Bahrain', 'Bahrain'), ('Canada',
'Canada'), ('Algeria', 'Algeria'), ('Belgium', 'Belgium'),
('Croatia', 'Croatia'), ('Kuwait', 'Kuwait'), ('Angola', 'Angola'),
('Greece', 'Greece'), ('Korea', 'Korea')])
```

Convert to a Lookup, using the initial letter of each country name as the key:

```
>>> countries = ['Austria', 'Bahrain', 'Canada', 'Algeria',
...              'Belgium', 'Croatia', 'Kuwait', 'Angola', 'Greece',
...              'Korea']
>>> query(countries).to_lookup(key_selector=lambda name: name[0])
Lookup([('A', 'Austria'), ('A', 'Algeria'), ('A', 'Angola'), ('B',
'Bahrain'), ('B', 'Belgium'), ('C', 'Canada'), ('C', 'Croatia'),
('K', 'Kuwait'), ('K', 'Korea'), ('G', 'Greece')])
```

Convert to a Lookup, using the initial letter of each country name as the key and the upper case name as the value:

```
>>> countries = ['Austria', 'Bahrain', 'Canada', 'Algeria',
...              'Belgium', 'Croatia', 'Kuwait', 'Angola', 'Greece',
...              'Korea']
>>> query(countries).to_lookup(key_selector=lambda name: name[0],
...                            value_selector=lambda name: name.upper())
Lookup([('A', 'AUSTRIA'), ('A', 'ALGERIA'), ('A', 'ANGOLA'), ('B',
'BAHRAIN'), ('B', 'BELGIUM'), ('C', 'CANADA'), ('C', 'CROATIA'),
('K', 'KUWAIT'), ('K', 'KOREA'), ('G', 'GREECE')])
```

to_set()

Convert the source sequence to a set.

Note: This method uses immediate execution.

Raises

- ValueError - If duplicate keys are in the projected source sequence.
- ValueError - If the Queryable is closed().

Examples

Convert a list to a set:

```
>>> a = [4, 9, 2, 3, 0, 1]
>>> query(a).to_set()
{0, 1, 2, 3, 4, 9}
```

Attempt to convert a list containing duplicates to a set:

```
>>> b = [6, 2, 9, 0, 2, 1, 9]
>>> query(b).to_set()
ValueError: Duplicate item value 2 in sequence during to_set()
```

to_str(*separator*)
Build a string from the source sequence.

The elements of the query result will each coerced to a string and then the resulting strings concatenated to return a single string. This allows the natural processing of character sequences as strings. An optional separator which will be inserted between each item may be specified.

Note: this method uses immediate execution.

Parameters **separator** – An optional separator which will be coerced to a string and inserted between each source item in the resulting string.

Returns A single string which is the result of stringifying each element and concatenating the results into a single string.

Raises

- TypeError - If any element cannot be coerced to a string.
- TypeError - If the separator cannot be coerced to a string.
- ValueError - If the Queryable is closed.

Examples

Convert a sequence of characters into a string:

```
>>> chars = ['c', 'h', 'a', 'r', 'a', 'c', 't', 'e', 'r', 's']
>>> query(chars).to_str()
'characters'
```

Concatenate some word fragments into a single string:

```
>>> syllables = ['pen', 'ta', 'syll', 'ab', 'ic']
>>> query(syllables).to_str()
```

Coerce some integers to strings and concatenate their digits to form a single string:

```
>>> codes = [72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100,␣
↪33]
>>> query(codes).to_str('-')
'72-101-108-108-111-44-32-87-111-114-108-100-33'
```

Coerce some integers to strings and concatenate their values separated by hyphens to form a single string:

```
>>> codes = [72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100,␣
↪33]
>>> query(codes).to_str('-')
'72-101-108-108-111-44-32-87-111-114-108-100-33'
```

to_tuple()
Convert the source sequence to a tuple.

Note: This method uses immediate execution.

Example

Convert from a list into a tuple:

```
>>> a = [1, 6, 8, 3, 4]
>>> query(a).to_list()
(1, 6, 8, 3, 4)
```

union (*second_iterable*, *selector=identity*)

Returns those elements which are either in the source sequence or in the second_iterable, or in both.

Note: This method uses deferred execution.

Parameters

- **second_iterable** – Elements from this sequence are returns if they are not also in the source sequence.
- **selector** – An optional single argument function which is used to project the elements in the source and second_iterables prior to comparing them. If omitted the identity function will be used.

Returns A sequence containing all elements in the source sequence and second sequence.

Raises

- ValueError - If the Queryable has been closed.
- TypeError - If the second_iterable is not in fact iterable.
- TypeError - If the selector is not callable.

Examples

Create a list of numbers which are in either or both of two lists:

```
>>> a = [1, 6, 9, 3]
>>> b = [2, 6, 7, 3]
>>> query(a).union(b).to_list()
[1, 6, 9, 3, 2, 7]
```

Create a list of numbers, based on their absolute values, which are in either or both of list `a` or list `b`, preferentially taking numbers from list `a` where the absolute value is present in both:

```
>>> a = [-1, -4, 2, 6, 7]
>>> b = [3, -4, 2, -6, 9]
>>> query(a).union(b, abs).to_list()
[-1, -4, 2, 6, 7, 3, 9]
```

where (*predicate*)

Filters elements according to whether they match a predicate.

Note: This method uses deferred execution.

Parameters **predicate** – A unary function which is applied to each element in the source sequence. Source elements for which the predicate returns True will be present in the result.

Returns A Queryable over those elements of the source sequence for which the predicate is True.

Raises

- ValueError - If the Queryable is closed.
- TypeError - If the predicate is not callable.

Example

Filter for elements greater than five:

```
>>> a = [1, 7, 2, 9, 3]
>>> query(a).where(lambda x: x > 5).to_list()
[7, 9]
```

zip(*second_iterable*, *result_selector=lambda x*, *y: (x*, *y)*)
Elementwise combination of two sequences.

The source sequence and the second iterable are merged element-by- element using a function to combine them into the single corresponding element of the result sequence. The length of the result sequence is equal to the length of the shorter of the two input sequences.

Note: This method uses deferred execution.

Parameters

- **second_iterable** – The second sequence to be combined with the source sequence.
- **result_selector** – An optional binary function for combining corresponding elements of the source sequences into an element of the result sequence. The first and second positional arguments are the elements from the source sequences. The result should be the result sequence element. If omitted, the result sequence will consist of 2-tuple pairs of corresponding elements from the source sequences.

Returns A Queryable over the merged elements.

Raises

- ValueError - If the Queryable is closed.
- TypeError - If result_selector is not callable.

Examples

Combine two sequences using the default result selector which creates a 2-tuple pair of corresponding elements:

```
>>> a = [1, 4, 6, 4, 2, 9, 1, 3, 8]
>>> b = [6, 7, 2, 9, 3, 5, 9]
>>> query(a).zip(b).to_list()
[(1, 6), (4, 7), (6, 2), (4, 9), (2, 3), (9, 5), (1, 9)]
```

Multiply the corresponding elements of two sequences to create a new sequence equal in length to the shorter of the two:

```
>>> a = [1, 4, 6, 4, 2, 9, 1, 3, 8]
>>> b = [6, 7, 2, 9, 3, 5, 9]
>>> query(a).zip(b, lambda x, y: x * y).to_list()
[6, 28, 12, 36, 6, 45, 9]
```

asq.queryables.OrderedQueryable

class asq.queryables.**OrderedQueryable**(*iterable*, *order*, *func*)

A Queryable representing an ordered iterable.

The sorting implemented by this class is an incremental partial sort so you don't pay for sorting results which are never enumerated.

—

TODO: Document OrderedQueryable

asq.queryables.Lookup

class asq.queryables.**Lookup**(*key_value_pairs*)

A multi-valued dictionary.

A Lookup represents a collection of keys, each one of which is mapped to one or more values. The keys in the Lookup are maintained in the order in which they were added. The values for each key are also maintained in order.

Note: Lookup objects are immutable.

All standard query operators may be used on a Lookup. When iterated or used as a Queryable the elements are returned as a sequence of Grouping objects.

—

Example

Lookup, being a subclass of Queryable supports all of the asq query operators over a collection of Groupings. For example, to select only those groups containing two or more elements and then flatten those groups into a single list, use:

```
>>> key_value_pairs = [('tree', 'oak'),
...                    ('bird', 'eagle'),
...                    ('bird', 'swallow'),
...                    ('tree', 'birch'),
...                    ('mammal', 'mouse'),
```

```
 ...                    ('tree', 'poplar')]
 ...
 >>> lookup = Lookup(key_value_pairs)
 >>> lookup.where(lambda group: len(group) >= 2).select_many().to_list()
['oak', 'birch', 'poplar', 'eagle', 'swallow']
```

__init__(*key_value_pairs*)

Construct a Lookup with a sequence of (key, value) tuples.

Parameters **key_value_pairs** – An iterable over 2-tuples each containing a key, value pair.

Example

To construct a Lookup from key value pairs:

```
>>> key_value_pairs = [('tree', 'oak'),
...                    ('bird', 'eagle'),
...                    ('bird', 'swallow'),
...                    ('tree', 'birch'),
...                    ('mammal', 'mouse'),
...                    ('tree', 'poplar')]
...
>>> lookup = Lookup(key_value_pairs)
```

__getitem__(*key*)

The sequence corresponding to a given key, or an empty sequence if there are no values corresponding to that key.

Parameters **key** – The key of the group to be returned.

Returns The Grouping corresponding to the supplied key.

Examples

To retrieve a Grouping for a given key:

```
>>> key_value_pairs = [('tree', 'oak'),
...                    ('bird', 'eagle'),
...                    ('bird', 'swallow'),
...                    ('tree', 'birch'),
...                    ('mammal', 'mouse'),
...                    ('tree', 'poplar')]
...
>>> lookup = Lookup(key_value_pairs)
>>> lookup['tree']
Grouping(key='tree')
```

but if no such key exists a Grouping will still be returned, albeit an empty one:

```
>>> vehicles = lookup['vehicle']
>>> vehicles
Grouping(key='vehicle')
>>> len(vehicles)
0
```

__len__()
Support for the len() built-in function.

Returns The number of Groupings (keys) in the lookup.

Example

To determine the number of Groupings in a Lookup:

```
>>> key_value_pairs = [('tree', 'oak'),
...                    ('bird', 'eagle'),
...                    ('bird', 'swallow'),
...                    ('tree', 'birch'),
...                    ('mammal', 'mouse'),
...                    ('tree', 'poplar')]
>>> lookup = Lookup(key_value_pairs)
>>> len(lookup)
3
```

__contains__()
Support for the 'in' membership operator.

Parameters **key** – The key for which membership will be tested.

Returns True if the Lookup contains a Grouping for the specified key, otherwise False.

Example

To determine whether a Lookup contains a specific Grouping:

```
>>> key_value_pairs = [('tree', 'oak'),
...                    ('bird', 'eagle'),
...                    ('bird', 'swallow'),
...                    ('tree', 'birch'),
...                    ('mammal', 'mouse'),
...                    ('tree', 'poplar')]
>>> lookup = Lookup(key_value_pairs)
>>> 'tree' in lookup
True
>>> 'vehicle' in lookup
False
```

__repr__()
Support for the repr() built-in function.

Returns The official string representation of the object.

Example

To produce a string representation of a Lookup:

```
>>> key_value_pairs = [('tree', 'oak'),
...                    ('bird', 'eagle'),
...                    ('bird', 'swallow'),
```

```
...                     ('tree', 'birch'),
...                     ('mammal', 'mouse'),
...                     ('tree', 'poplar')]
...
>>> lookup = Lookup(key_value_pairs)
>>> repr(lookup)
"Lookup([('tree', 'oak'), ('tree', 'birch'), ('tree', 'poplar'),
('bird', 'eagle'), ('bird', 'swallow'), ('mammal', 'mouse')])"
```

apply_result_selector(*selector*)

Example

Convert each group to a set using a lambda selector and put the resulting sets in a list:

```
>>> key_value_pairs = [('tree', 'oak'),
...                    ('bird', 'eagle'),
...                    ('bird', 'swallow'),
...                    ('tree', 'birch'),
...                    ('mammal', 'mouse'),
...
>>> lookup = Lookup(key_value_pairs)
>>> lookup.apply_result_selector(lambda key, group: set(group)).to_
↪list()
[set(['poplar', 'oak', 'birch']), set(['eagle', 'swallow']),
set(['mouse'])]
```

to_dictionary(*key_selector=None*, *value_selector=None*)

Build a dictionary from the source sequence.

Parameters

- **key_selector** – A unary callable to extract a key from each item. By default the key of the Grouping.
- **value_selector** – A unary callable to extract a value from each item. By default the value is the list of items from the Grouping.

Note: This method uses immediate execution.

Raises

- ValueError - If the Queryable is closed.
- TypeError - If key_selector is not callable.
- TypeError - If value_selector is not callable.

Example

Convert a `Lookup` to a `dict` using the default selectors which produce a dictionary mapping the lookup keys to lists:

```
>>> key_value_pairs = [('tree', 'oak'),
...                    ('bird', 'eagle'),
...                    ('bird', 'swallow'),
...                    ('tree', 'birch'),
...                    ('mammal', 'mouse'),
...
>>> lookup = Lookup(key_value_pairs)
>>> lookup.to_dictionary()
{'mammal': ['mouse'], 'bird': ['eagle', 'swallow'], 'tree': ['oak',
↪'birch']}
```

Providing a `value_selector` to construct the values of the dictionary as a `set` rather than the default `list`:

```
>>> lookup.to_dictionary(value_selector=set)
{'mammal': {'mouse'}, 'bird': {'swallow', 'eagle'}, 'tree': {'birch',
↪ 'oak'}}
```

asq.queryables.Grouping

class asq.queryables.**Grouping**(*key*, *items*)

A collection of objects which share a common key.

All standard query operators may be used on a Grouping.

Note: It is not intended that clients should directly create Grouping objects. Instances of this class are retrieved from Lookup objects.

Example

Grouping, being a subclass of Queryable, supports all of the `asq` query operators. For example, to produce a list of the group items in upper case:

```
>>> g = Grouping("fruit", ["pear", "apple", "orange", "banana"])
>>> g.select(str.upper).to_list()
['PEAR', 'APPLE', 'ORANGE', 'BANANA']
```

__init__(*key*, *iterable*)

Create a Grouping with a given key and a collection of members.

Parameters

- **key** – The key corresponding to this Grouping
- **items** – An iterable collection of the members of the group.

Example

Construct a Grouping from a list:

```
>>> Grouping("fruit", ["pear", "apple", "orange", "banana"])
Grouping(key='fruit')
```

key
The key common to all elements.

Example

To retrieve the key from a Grouping:

```
>>> g = Grouping("fruit", ["pear", "apple", "orange", "banana"])
>>> g.key
'fruit'
```

__len__()
The number of items in the Grouping.

Example

To retrieve the number of items in a Grouping:

```
>>> g = Grouping("fruit", ["pear", "apple", "orange", "banana"])
>>> len(g)
4
```

__eq__()
Determine value equality with another grouping.

Parameters **rhs** – The object on the right-hand-side of the comparison must support a property called 'key' and be iterable.

Returns True if the keys and sequences are equal, otherwise False.

Example

To test whether two Groupings are equal in value:

```
>>> g1 = Grouping("fruit", ["pear", "apple", "orange", "banana"])
>>> g2 = Grouping("fruit", ["pear", "apple", "orange", "banana"])
>>> g1 == g2
True
```

__ne__()
Determine value inequality with another grouping.

Parameters **rhs** – The object on the right-hand-side of the comparison must support a property called 'key' and be iterable.

Returns True if the keys or sequences are not equal, otherwise False.

Example

To test whether two Groupings are inequal in value:

```
>>> g1 = Grouping("fruit", ["pear", "apple", "orange", "banana"])
>>> g2 = Grouping("fruit", ["cherry", "apple", "orange", "banana"])
>>> g1 != g2
True
```

__repr__()

Example

To create a string representation of the Grouping:

```
>>> g = Grouping("fruit", ["pear", "apple", "orange", "banana"])
>>> repr(g)
Grouping(key="fruit", items=["pear", "apple", "orange", "banana"])
```

to_dictionary(*key_selector=None*, *value_selector=None*)
Build a dictionary from the source sequence.

Parameters

- **key_selector** – A unary callable to extract a key from each item or None. If None, the default key selector produces a single dictionary key, which if the key of this Grouping.
- **value_selector** – A unary callable to extract a value from each item. If None, the default value selector produces a list, which contains all elements from this Grouping.

Note: This method uses immediate execution.

Raises

- ValueError - If the Queryable is closed.
- TypeError - If key_selector is not callable.
- TypeError - If value_selector is not callable.

Examples

Convert a `Grouping` to a `dict` using the default selectors:

```
>>> g = Grouping("fruit", ["pear", "apple", "orange", "banana"])
>>> g.to_dictionary()
{'fruit': ['pear', 'apple', 'orange', 'banana']}
```

Providing a `key_selector` and to generate the dictionary keys from the length of each element in the `Grouping`:

```
>>> g.to_dictionary(key_selector=len, value_selector=identity)
{4: 'pear', 5: 'apple', 6: 'banana'}
```

Notice that first six-letter word ‘orange’ is overwritten by the second six-letter word, ‘banana’.

Since the key of the `Grouping` is not availble via the items in the collection, if you need to incorporate the key into the produced `dict` it must be incorporated into the selectors:

```
>>> g.to_dictionary(
...     key_selector=lambda item: '{} letter {}'.format(len(item), g.
↪key),
...     value_selector=str.capitalize)
...
{'5 letter fruit': 'Apple', '6 letter fruit': 'Banana', '4 letter␣
↪fruit': 'Pear'}
```

asq.selectors

Selector functions and selector function factories.

Selectors are so-called because they are used to select a value from an element. The selected value is often an attribute or sub-element but could be any computed value. The `selectors` module provides to standard selectors and also some selector factories.

Selectors

identity	The identity function.

asq.selectors.**identity**(*x*)

The identity function.

The identity function returns its only argument.

Parameters **x** – A value that will be returned.

Returns The argument x.

Examples

Use the the identity function with the `where()` query operator, which has the effect that only elements which evaluate to True are present in the result:

```
>>> from selectors import identity
>>> a = [5, 3, 0, 1, 0, 4, 2, 0, 3]
>>> query(a).where(identity).to_list()
[5, 3, 1, 4, 2, 3]
```

Selector factories

*a_*	alias of `attrgetter`
*k_*	alias of `itemgetter`
*m_*	alias of `methodcaller`

asq.selectors.**a_**(*name*)

attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

Longhand equivalent

The selector factory call:

```
a_(name)
```

is equivalent to the longhand:

```
lambda element: element.name
```

Example

From a list of spaceship characteristics order the spaceships by length and select the spaceship name:

```
>>> from asq.selectors import a_
>>> class SpaceShip(object):
...     def __init__(self, name, length, crew):
...         self.name = name
...         self.length = length
...         self.crew = crew
...
>>> spaceships = [SpaceShip("Nebulon-B", 300, 854),
...               SpaceShip("V-19 Torrent", 6, 1),
...               SpaceShip("Venator", 1137, 7400),
...               SpaceShip("Lambda-class T-4a shuttle", 20, 6),
...               SpaceShip("GR-45 medium transport", 90, 6)]
>>> query(spaceships).order_by(a_('length')).select(a_('name')).to_list()
['V-19 Torrent', 'Lambda-class T-4a shuttle', 'GR-45 medium transport',
 'Nebulon-B', 'Venator']
```

or sort the

asq.selectors.**k_**(*key*)
itemgetter(item, ...) –> itemgetter object

Return a callable object that fetches the given item(s) from its operand. After f = itemgetter(2), the call f(r) returns r[2]. After g = itemgetter(2, 5, 3), the call g(r) returns (r[2], r[5], r[3])

Longhand equivalent

The selector factory call:

```
k_(key)
```

is equivalent to the longhand:

```
lambda element: element[name]
```

Example

From a list of dictionaries containing planetary data, sort the planets by increasing mass and select their distance from the sun:

```
>>> from asq.selectors import k_
>>> planets = [dict(name='Mercury', mass=0.055, period=88),
...            dict(name='Venus', mass=0.815, period=224.7),
...            dict(name='Earth', mass=1.0, period=365.3),
...            dict(name='Mars', mass=0.532, period=555.3),
...            dict(name='Jupiter', mass=317.8, period=4332),
...            dict(name='Saturn', mass=95.2, period=10761),
...            dict(name='Uranus', mass=14.6, period=30721),
...            dict(name='Neptune', mass=17.2, period=60201)]
>>> query(planets).order_by(k_('mass')).select(k_('period')).to_list()
[88, 555.3, 224.7, 365.3, 30721, 60201, 10761, 4332]
```

asq.selectors.**m_**(*name*, **args*, ***kwargs*)
methodcaller(name, ...) –> methodcaller object

Return a callable object that calls the given method on its operand. After f = methodcaller('name'), the call f(r) returns r.name(). After g = methodcaller('name', 'date', foo=1), the call g(r) returns r.name('date', foo=1).

Longhand equivalent

The selector factory call:

```
m_(name, *args, **kwargs)
```

is equivalent to the longhand:

```
lambda element: getattr(element, name)(*args, **kwargs)
```

Example

From a list of SwimmingPool objects compute a list of swimming pool areas by selecting the `area()` method on each pool:

```
>>> class SwimmingPool(object):
...     def __init__(self, length, width):
...         self.length = length
...         self.width = width
...     def area(self):
...         return self.width * self.length
...     def volume(self, depth):
...         return self.area() * depth
...
>>> pools = [SwimmingPool(50, 25),
...          SwimmingPool(25, 12.5),
...          SwimmingPool(100, 25),
...          SwimmingPool(10, 10)]
>>> query(pools).select(m_('area')).to_list()
[1250, 312.5, 2500, 100]
```

Compute volumes of the above pools for a water depth of 2 metres by passing the depth as a positional argument to the *m_()* selector factory:

```
>>> query(pools).select(m_('volume', 2)).to_list()
[2500, 625.0, 5000, 200]
```

Alternatively, we can use a named parameter to make the code clearer:

```
>>> query(pools).select(m_('volume', depth=1.5)).to_list()
[1875.0, 468.75, 3750.0, 150.0]
```

asq.predicates

Predicate function factories

Predicates are boolean functions which return True or False.

Predicate factories

The predicate factories partially apply the binary comparison operators by providing the right-hand-side argument. The result is a unary function the single argument to which is the left-hand-side of the comparison operator.

For example. the `lt_(rhs)` predicate factory returns:

```
lambda lhs: lhs < rhs
```

where `rhs` is provided when the predicate is created but `lhs` takes the value passed to the unary predicate.

*contains_*	Create a unary predicate which tests for membership if its argument.
*eq_*	Create a predicate which tests its argument for equality with a value.
*is_*	Create a predicate which performs an identity comparison of its argument with a value.
*ge_*	Create a predicate which performs a greater-than-or-equal comparison of its argument with a value.
*gt_*	Create a predicate which performs a greater-than comparison of its argument with a value.
*le_*	Create a predicate which performs a less-than-or-equal comparison of its argument with a value.
*lt_*	Create a predicate which performs a less-than comparison of its argument with a value.
*ne_*	Create a predicate which tests its argument for inequality with a value.

asq.predicates.**contains_**(*lhs*)

Create a unary predicate which tests for membership if its argument.

Parameters **lhs** – (left-hand-side) The value to test for membership for in the predicate argument.

Returns A unary predicate function which determines whether its single arguments (lhs) contains lhs.

Example

Filter for specific words containing 'ei':

```
>>> words = ['banana', 'receive', 'believe', 'ticket', 'deceive']
>>> query(words).where(contains_('ei')).to_list()
['receive', 'deceive']
```

asq.predicates.**eq_**(*rhs*)

Create a predicate which tests its argument for equality with a value.

Parameters **rhs** – (right-hand-side) The value with which the left-hand-side element will be compared for equality.

Returns A unary predicate function which compares its single argument (lhs) for equality with rhs.

Example

Filter for those numbers equal to five:

```
>>> numbers = [5, 9, 12, 5, 89, 34, 2, 67, 43]
>>> query(numbers).where(eq_(5)).to_list()
[5, 5]
```

asq.predicates.**is_**(*rhs*)

Create a predicate which performs an identity comparison of its argument with a value.

Parameters **rhs** – (right-hand-side) The value against which the identity test will be performed.

Returns A unary predicate function which determines whether its single arguments (lhs) has the same identity - that is, is the same object - as rhs.

Example

Filter for a specific object by identity:

```
>>> sentinel = object()
>>> sentinel
<object object at 0x0000000002ED8040>
>>> objects = ["Dinosaur", 5, sentinel, 89.3]
>>> query(objects).where(is_(sentinel)).to_list()
[<object object at 0x0000000002ED8040>]
>>>
```

asq.predicates.**ge_**(*rhs*)

Create a predicate which performs a greater-than-or-equal comparison of its argument with a value.

Parameters **rhs** – (right-hand-side) The value against which the greater-than-or- equal test will be performed.

Returns A unary predicate function which determines whether its single argument (lhs) is greater-than rhs.

Example

Filter for those numbers greater-than-or-equal to 43:

```
>>> numbers = [5, 9, 12, 5, 89, 34, 2, 67, 43]
>>> query(numbers).where(ge_(43)).to_list()
[89, 67, 43]
```

asq.predicates.**gt_**(*rhs*)

Create a predicate which performs a greater-than comparison of its argument with a value.

Parameters **rhs** – (right-hand-side) The value against which the greater-than test will be performed.

Returns A unary predicate function which determines whether its single argument (lhs) is less-than-or-equal to rhs.

Example

Filter for those numbers greater-than 43:

```
>>> numbers = [5, 9, 12, 5, 89, 34, 2, 67, 43]
>>> query(numbers).where(gt_(43)).to_list()
[89, 67]
```

asq.predicates.**le_**(*rhs*)

Create a predicate which performs a less-than-or-equal comparison of its argument with a value.

Parameters **rhs** – (right-hand-side) The value against which the less-than-or-equal test will be performed.

Returns A unary predicate function which determines whether its single argument (lhs) is less-than-or-equal to rhs.

Example

Filter for those numbers less-than-or-equal to 43:

```
>>> numbers = [5, 9, 12, 5, 89, 34, 2, 67, 43]
>>> query(numbers).where(le_(43)).to_list()
[5, 9, 12, 5, 34, 2, 43]
```

asq.predicates.**lt_**(*rhs*)

Create a predicate which performs a less-than comparison of its argument with a value.

Parameters **rhs** – (right-hand-side) The value against which the less-than test will be performed.

Returns A unary predicate function which determines whether its single argument (lhs) is less-than rhs.

Example

Filter for those numbers less-than-or-equal to 43:

```
>>> numbers = [5, 9, 12, 5, 89, 34, 2, 67, 43]
>>> query(numbers).where(lt_(43)).to_list()
[5, 9, 12, 5, 34, 2]
```

asq.predicates.**ne_**(*rhs*)

Create a predicate which tests its argument for inequality with a value.

Parameters **rhs** – (right-hand-side) The value with which the left-hand-side element will be compared for inequality.

Returns A unary predicate function which compares its single argument (lhs) for inequality with rhs.

Example

Filter for those numbers not equal to 5:

```
>>> numbers = [5, 9, 12, 5, 89, 34, 2, 67, 43]
>>> query(numbers).where(ne_(5)).to_list()
[9, 12, 89, 34, 2, 67, 43]
```

Predicate combinators

Predicate combinators allow the predicate factories to be modified and combined in a concise way. For example, we can write:

```
or_(lt_(5), gt_(37))
```

which will produce a predicate equivalent to:

```
lambda lhs: lhs < 5 or lhs > 37
```

which can be applied to each element of a sequence to test whether the element is outside the range 5 to 37.

*and_*	A predicate combinator which produces the a new predicate which is the logical conjunction of two existing unary predicates.
*not_*	A predicate combinator which negates produces an inverted predicate.
*or_*	A predicate combinator which produces the a new predicate which is the logical disjunction of two existing unary predicates.
*xor_*	A predicate combinator which produces the a new predicate which is the logical exclusive disjunction of two existing unary predicates.

asq.predicates.**and_**(*predicate1*, *predicate2*)

A predicate combinator which produces the a new predicate which is the logical conjunction of two existing unary predicates.

The predicate returned by this combinator returns True when both of the two supplied predicates return True, otherwise it returns False.

Parameters

- **predicate1** – A unary predicate function.
- **predicate2** – A unary predicate function.

Returns A unary predicate function which is the logical conjunction of predicate1 and predicate2.

..rubric:: Example

Filter a list for all the words which both start with 'a' and end 't':

```
>>> words = ['alphabet', 'train', 'apple', 'aghast', 'tent', 'alarm']
>>> query(words).where(and_(m_('startswith', 'a'), m_('endswith', 't'))).
↪to_list()
['alphabet', 'aghast']
```

asq.predicates.**not_**(*predicate*)

A predicate combinator which negates produces an inverted predicate.

The predicate returned by this combinator is the logical inverse of the supplied combinator.

Parameters **predicate** – A unary predicate function to be inverted.

Returns A unary predicate function which is the logical inverse of pred.

Example

Filter a list for all the word which do not contain a specific sentinel object:

```
>>> sentinel = object()
>>> objects = ["Dinosaur", 5, sentinel, 89.3]
>>> query(objects).where(not_(is_(sentinel))).to_list()
['Dinosaur', 5, 89.3]
```

asq.predicates.**or_**(*predicate1*, *predicate2*)

A predicate combinator which produces the a new predicate which is the logical disjunction of two existing unary predicates.

The predicate returned by this combinator returns True when either or both of the two supplied predicates return True, otherwise it returns False.

Parameters

- **predicate1** – A unary predicate function.
- **predicate2** – A unary predicate function.

Returns A unary predicate function which is the logical disjunction of predicate1 and predicate2.

Example

Filter a list for all words which either start with 'a' or end with 't':

```
>>> words = ['alphabet', 'train', 'apple', 'aghast', 'tent', 'alarm']
>>> query(words).where(or_(m_('startswith', 'a'), m_('endswith', 't'))).
↪to_list()
['alphabet', 'apple', 'aghast', 'tent', 'alarm']
```

asq.predicates.**xor_**(*predicate1*, *predicate2*)
A predicate combinator which produces the a new predicate which is the logical exclusive disjunction of two existing unary predicates.

The predicate returned by this combinator returns True when the two supplied predicates return the different values, otherwise it returns False.

Parameters

- **predicate1** – A unary predicate function.
- **predicate2** – A unary predicate function.

Returns A unary predicate function which is the logical exclusive disjunction of predicate1 and predicate2.

Example

Filter a list for all words which either start with 'a' or end with 't' but not both:

```
>>> words = ['alphabet', 'train', 'apple', 'aghast', 'tent', 'alarm']
>>> query(words).where(xor_(m_('startswith', 'a'), m_('endswith', 't'))).
↪to_list()
['apple', 'tent', 'alarm']
```

asq.record

Records provide a convenient anonymous class which can be useful for managing intermediate query results. `new()` provides a concise way to create `Records` in the middle of a query.

asq.record.Record

class asq.record.**Record**(***kwargs*)
A class to which any attribute can be added at construction.

__init__(***kwargs*)
Initialise a Record with an attribute for each keyword argument.

The attributes of a Record are mutable and may be read from and written to using regular Python instance attribute syntax.

Parameters ****kwargs** – Each keyword argument will be used to initialise an attribute with the same name as the argument and the given value.

__repr__()
A valid Python expression string representation of the Record.

__str__()
A string representation of the Record.

asq.record.new

asq.record.**new**(***kwargs*)
A convenience factory for creating Records.

Parameters ****kwargs** – Each keyword argument will be used to initialise an attribute with the same name as the argument and the given value.

Returns A Record which has a named attribute for each of the keyword arguments.

Example

Create an employee and the get and set attributes:

```
>>> employee = new(age=34, sex='M', name='Joe Bloggs', scores=[3, 2, 9,␣
↪8])
>>> employee
Record(age=34, scores=[3, 2, 9, 8], name='Joe Bloggs', sex='M')
>>> employee.age
34
>>> employee.name
'Joe Bloggs'
>>> employee.age = 35
>>> employee.age
35
```

asq.namedelements

This module contains the definition of the IndexedElement type.

`IndexedElements` and `KeyedElement` are `namedtuples` useful for storing index, element pairs. They are used as the default selectors by the `select_with_index()` and `select_many_with_index()`, `select_with_correspondence()` and `select_many_with_corresponding()` query methods.

asq.namedelements.IndexedElement

class `asq.namedelements.`**IndexedElement**(*index*, *value*)

The index and value of the element can be accessed via the `index` and `value` attributes.

static **__new__**(*index*, *value*)

Create new instance of IndexedElement(index, value)

__repr__()

Return a nicely formatted representation string

__str__()

x.__str__() <==> str(x)

asq.namedelements.KeyedElement

class `asq.namedelements.`**KeyedElement**(*key*, *value*)

The key and associated value can be accessed via the `key` and `value` attributes.

static **__new__**(*key*, *value*)

Create new instance of KeyedElement(key, value)

__repr__()

Return a nicely formatted representation string

__str__()
x.__str__() <==> str(x)

asq.extension

Adding extension operators.

The `extension` modules contains tools for registering new extension operators with `asq`. This is achieved by dynamically adding new methods to `Queryable` and possibly its subclasses.

add_method	Add an existing function to a class as a method.
extend	A function decorator for extending an existing class.

asq.extension.**add_method**(*function*, *klass*, *name=None*)
Add an existing function to a class as a method.

Note: Consider using the extend decorator as a more readable alternative to using this function directly.

Parameters

- **function** – The function to be added to the class klass.
- **klass** – The class to which the new method will be added.
- **name** – An optional name for the new method. If omitted or None the original name of the function is used.

Returns The function argument unmodified.

Raises

ValueError - If klass already has an attribute with the same name as the extension method.

Example

Define a function called `every_second()` which returns every second element from the source and add it to `Queryable` as a new query operator called `alternate()`:

```
>>> def every_second(self):
...     def generate():
...         for index, item in enumerate(self):
...             if index % 2 == 0:
...                 yield item
...     return self._create(generate())
...
>>> from asq.extension import add_method
>>> from asq.queryables import Queryable
>>>
>>> add_method(every_second, Queryable, "alternate")
<function every_second at 0x0000000002D2D5C8>
>>> a = [5, 8, 3, 2, 0, 9, 5, 4, 9, 2, 7, 0]
>>> query(a).alternate().to_list()
[5, 3, 0, 5, 9, 7]
```

asq.extension.**extend**(*klass*, *name=None*)

A function decorator for extending an existing class.

Use as a decorator for functions to add to an existing class.

Parameters

- **klass** – The class to be decorated.
- **name** – The name the new method is to be given in the klass class.

Returns A decorator function which accepts a single function as its only argument. The decorated function will be added to class klass.

Raises

ValueError - If klass already has an attribute with the same name as the extension method.

Example

Define a new query method called `pairs()` which iterates over successive pairs in the source iterable, add it to the `Queryable` class and use it to execute a query. Note that extension methods defined in this way will typically need to call internal methods of `Queryable`, such as the `_create()` method used here to construct a new `Queryable`:

```
>>> from asq.extension import extend
>>> from asq.queryables import Queryable
>>>
>>> @extend(Queryable)
... def pairs(self):
...     def generate_pairs():
...         i = iter(self)
...         sentinel = object()
...         prev = next(i, sentinel)
...         if prev is sentinel:
...             return
...         for item in i:
...             yield prev, item
...             prev = item
...     return self._create(generate_pairs())
...
>>> from asq import query
>>> a = [5, 4, 7, 2, 8, 9, 1, 0, 4]
>>> query(a).pairs().to_list()
[(5, 4), (4, 7), (7, 2), (2, 8), (8, 9), (9, 1), (1, 0), (0, 4)]
```

1.3.2 Differences from LINQ

Although `asq` is inspired by LINQ, there are inevitably some differences with Microsoft's LINQ on .NET in order to accommodate the variance between C# and Python.

Embedded Domain Specific Language

C# and VB.NET have specific syntax extensions to support the creation of LINQ queries. This provides an alternative to the fluent interface (method chaining). Any LINQ query can be expressed using the fluent interface. This is not

true for the LINQ domain specific languages embedded in C# and VB.NET but they provide syntactic sugar for many common queries structures.

For example the following LINQ comprehension expression in C#:

```
from item in collection where item.id == 3 select item
```

is equivalent to the following call without syntactic sugar in C#:

```
collection.Where(item => item.id == 3)
```

No language extensions are provided by `asq`; however, the fluent interface is fully supported.

`let` bindings

LINQ query syntax includes a `let` keyword which has no direct equivalent in the LINQ fluent (method chaining) interface. The `let` keyword introduces a new identifier which can store intermediate query results for improvements in readability or performance.

All queries in LINQ syntax are translated by the C# compiler into chained method calls. The `let` keyword is translated into a `select()` mapping which creates instances of anonymous types which bundle together the current query value together with any addition values bound by `let` so they may all be passed down the method chain. Selectors and predicates in the method chain following the `select()` are modified to extract the correct members from the anonymous type.

For example, the following LINQ query expression:

```
var names = new string[] { "Dog", "Cat", "Giraffe", "Monkey", "Tortoise" };
var result =
    from animalName in names
    let nameLength = animalName.Length
    where nameLength > 3
    orderby nameLength
    select animalName;
```

is equivalent to the C# method chain:

```
var result = names
    .Select(animalName => new { nameLength = animalName.Length, animalName})
    .Where(x=>x.nameLength > 3)
    .OrderBy(x=>x.nameLength)
    .Select(x=>x.animalName);
```

The latter form can be emulated in `asq` using a `Record` object which can be concisely created by the `new()` factory function:

```
from asq.initiators import asq
from asq.record import new

names = ['Dog', 'Cat', 'Giraffe', 'Monkey', 'Tortoise']
result = query(names)
    .select(lambda animal_name: new(name_length=len(animal_name),
                                    animal_name=animal_name))
    .where(lambda x: x.name_length > 3)
    .order_by(lambda x: x.name_length)
    .select(lambda x: x.animal_name)
```

Extension methods

C# supports extension methods which allow LINQ to "add" methods to existing types such as `IEnumerable`. This is how the LINQ query operators are added to enumerable types. Python has no fully equivalent technique because so-called monkey patching, whereby new methods can be added to existing classes, cannot be applied to built-in types such as list because they are immutable by design.

For this reason query initiators such `query()` must be used to convert a Python iterable into a type which supports query operators.

Nonetheless, the core operators included in `asq` may be supplemented with additional operators by adding new methods to the appropriate queryable type, usually `Queryable` itself.

A decorator called `@extend` is provided by `asq` for this purpose.

Overloading

Being statically typed C# supports method overloading and this is used extensively by LINQ. For example, the `SelectMany()` method has the following four overloads:

```
SelectMany<TSource, TResult>(IEnumerable<TSource>,
                             Func<TSource, IEnumerable<TResult>>)

SelectMany<TSource, TResult>(IEnumerable<TSource>,
                             Func<TSource, Int32, IEnumerable<TResult>>)

SelectMany<TSource, TCollection, TResult>(IEnumerable<TSource>,
                                          Func<TSource, IEnumerable<TCollection>>,
                                          Func<TSource, TCollection, TResult>)

SelectMany<TSource, TCollection, TResult>(IEnumerable<TSource>,
                                          Func<TSource, Int32, IEnumerable
↪<TCollection>>,
                                          Func<TSource, TCollection, TResult>)
```

These four overloads perform quite distinct, although related, operations. In `asq` the equivalent of these overloads are methods with separate - and more descriptive - names:

```
select_many(collection_selector, result_selector)

select_many_with_index(collection_selector, result_selector)

select_many_with_correspondence(collection_selector, result_selector)
```

Default arguments allow the Python `select_many()` method to perform the equivalent function as the first and third C# overloads and `select_many_with_index()` the second and fourth overloads. The third Python method provides a simpler alternative to the second version in some scenarios.

Equality comparers

Many .NET containers and and LINQ operators allow the specification of comparer objects, particularly IEqualityComparer. This is important in C# because equality in C# using the equality operator is by reference rather than value. The use of separate comparer types is not idiomatic in Python and in general no attempt has been made to support the equivalent of LINQ operator overloads which accept equality comparers.

Two `asq` operators which *do* accept equality comparison functions are `contains()` and `sequence_equal()`.

Style changes

All class and method names in `asq` are compatible with the PEP 8 style- guide. This necessarily requires that they are different to the .NET methods, so, for example, `SelectMany()` in .NET becomes `select_many()` in `asq`.

The LINQ IEnumerable extension methods which create new sequences rather than operate on existing sequences have become module-scope free function *initiators* in `asq` in the `initiators` sub-module.

Specific naming changes

Owing to clashes with existing Python types, some specific name changes have been made. Other name changes have been made because overloads in LINQ have become separate named methods in `asq`.

LINQ	*asq*
`IEnumerable`	`query(iterable)`
`range()`	`integers()`
`except()`	`difference()`

Selector and predicate factories

Lambdas in Python are relatively verbose compared to C# lambdas and have the further restriction that they cannot span multiple lines. Selector and predicate factories are provided to `asq` to generate common lambda forms. These have some out-of-the-box equivalent in LINQ.

Execution engine

The LINQ implementation in .NET converts query expressions or method chains into an abstract representation of the query in the form of expression trees. This allows decoupling of query specification from the form of the which will be queried. This allows queries to be applied to diverse data sources including object sequences as represented by IEnumerable (LINQ-to-objects), database (LINQ-to-SQL), XML (LINQ-to-XML) or indeed any other data source for which a LINQ provider has been created.

At this stage in it's development `asq` sets out to be a solid, Pythonic, functional equivalent of LINQ-to-objects only. With only one data provider there is not advantage to representing queries in some abstract intermediate representation. An expression tree based implementation of `asq` may be created in future.

Pythonic behaviour

Container creation

Included in `asq` are several additions to support idiomatic Python usage. The first group are the `to_*()` methods where * is a placeholder for various built-in types (`list`, `set`, `dict`, `tuple`) and `asq` provided types (`lookup`).

Special methods

The following Python special methods are supported by the `Queryable` type to support idiomatic Python usage.

Special method	Purpose	Equivalent query operator
`__contains__`	Support for the `in` operator	`contains()`
`__getitem__`	Support for indexing with []	`element_at()`
`__reversed__`	Support for reversed() built-in	`reverse()`
`__repr__`	Stringified representation	
`__str__`	Stringified representation	

So, for example, the expression:

```
5 in query(numbers).select(lambda: x * 2)
```

is equivalent to:

```
query(numbers).select(lambda: x * 2).contains(5)
```

1.3.3 Frequently Asked Questions

Where are `map()`, `filter()` and `reduce()`?

All three of these operators exist in `asq` with different spelling for consistency with LINQ:

Python Standard Libary	asq
`map()`	`select()`
`filter()`	`where()`
`reduce()`	`aggregate()`

Where are `fold()`, `foldl()` and `foldr()`?

Folds in `asq` can be performed using `aggregate()`. Here are the equivalents of some Haskell code using folds and the Python `asq` code using `aggregate()`:

Haskell	asq
`foldl f seed seq`	`query(seq).aggregate(f, seed)`
`foldr f seed seq`	`query(seq).reverse().aggregate(f, seed)`

Wouldn't generators be a better name for what `asq` calls initiators?

Possibly, but it could be confused with other uses of the word 'generator' in Python. In fact, `asq`'s initiators might actually *be* generators but the essential point is that they 'initiate' the fluent query interface of `asq`.

How do I pronounce `asq`?

See the answer to the next question.

Where does the name `asq` come from?

Well, "asq" is homophonic with "ask" which is in turn synonymous with "query". Further more, "query" contains a "q" which rather neatly takes us back to the "q" in "asq". The inspiration for `asq` comes from "LINQ" where the "Q" also stands for "query". Finally, the glyph "q" is mirror symmetric with "p" and replacing the "q" in "asq" with "p" gives "asp" which also rhymes with "asq" but more importantly is synonymous with "snake". "asq" is written in Python, and pythons are a kind of snake, although the programming language is actually named after a popular British comedy troupe and nothing to do with snakes at all. Or something.

1.4 Detailed Change History

1.4.1 Changes

asq v.next

- Adds a convenience alias for `asq.initiators.query` as `asq.query`.

asq 1.3

There are several minor breaking API changes in this release. Please read carefully more details:

- Re-assigns copyright from Robert Smallshire to Sixty North AS.
- Adds `select_with_correspondence()` query method.
- Renames the `indexedelement` module to `namedelements`.
- Renames the second element of `IndexedElement` from `element` to `value`.
- Adds the `KeyedElement` `namedtuple` to the `namedelements` module. `KeyedElement` has two elements called `key` and `value`.
- Queryable.to_dictionary() no longer raises an exception if the key_selector produces duplicate keys. Instead, the values associated with later keys overwrite those produced by earlier keys. This weakening of the to_dictionary() constract allows us to maintain Liskov subsstitutability in light of the specialised default key and value selectors for the overrides of `to_dictionary()` provided for the `Lookup` and `Grouping` classes. (See the next two changes for more details).
- Less surprising behaviour for `Lookup.to_dictionary()`: The default key and value selectors for `Lookup.to_dictionary()` are overidden, so that the produced dictionary contains a single item for each `Grouping` such that the key of each item is the key of the corresponding `Grouping` and the value of the item is a list of the elements from the `Grouping`.
- Less surprising behaviour for `Grouping.to_dictionary()`: The default key and value selectors for `Grouping.to_dictionary()` are overidden, so that the produced dictionary contains a single item, such that the key of the item is the key of the `Grouping` and the value of the item is a `list` containing the elements from the `Grouping`.

asq 1.2.1

- Fixes a problem in setup.py that prevented installation on Python 2.

asq 1.2

- The default selector for select_with_index() now produces a new IndexedElement object for each type which is a namedtuple. As IndexedElement is a tuple this change is backwards compatibile, but now the more readable item.index and item.element attributes can be used instead of accessing via indexes zero and one.

asq 1.1

- The selector factories k_(), a_() and m_() have much faster implementations because they are now simply aliases for itemgetter, attrgetter and methodcaller from the Python standard library operator module. As a result,

even though they remain backwards API compatible with those in asq 1.0 their capabilities are also extended somewhat:

  - **k_** can optionally accept more than one argument (key) and if so, the selector it produces will return a tuple of multiple looked-up values rather than a single value.
  - **a_** can optionally accept more than one argument (key) and if so, the selector it produces will return a tuple of multiple looked-up values rather than a single value. Furthermore, the attribute names supplied in each argument can now contain dots to refer to nested attributes.

- Added asq.selectors.make_selector which will create a selector directly from a string or integer using attribute or item lookup respectively.

asq 1.0

Huge correctness and completeness changes for 1.0 since 0.9. The API now has feature equivalence with LINQ for objects with 100% test coverage and complete documentation.

The API has been very much reorganised with some renaming of crucial functions. The important asq() function is now called query() to prevent a clash with the package name itself and is found in the asq.initiators package.

For common asq usage you now need to do:

```
from asq.initiators import query
a = [1, 2, 3]
query(a).select(lambda x: x*x).to_list()
```

to get started. For more than that, consult the documentation.

1.5 Samples

More complex examples of non-trivial usage of `asq`:

1.5.1 Samples

Mandelbrot

Visualising the Mandelbrot fractal with `asq`. This is a direct translation of Jon Skeet's original LINQ Mandelbrot from LINQ in C# to `asq` in Python. The sample requires the Python Imaging Library and so at the time of writing only works with Python 2.

This example can be found in the source distribution of `asq` under `asq/examples/mandelbrot.py`.

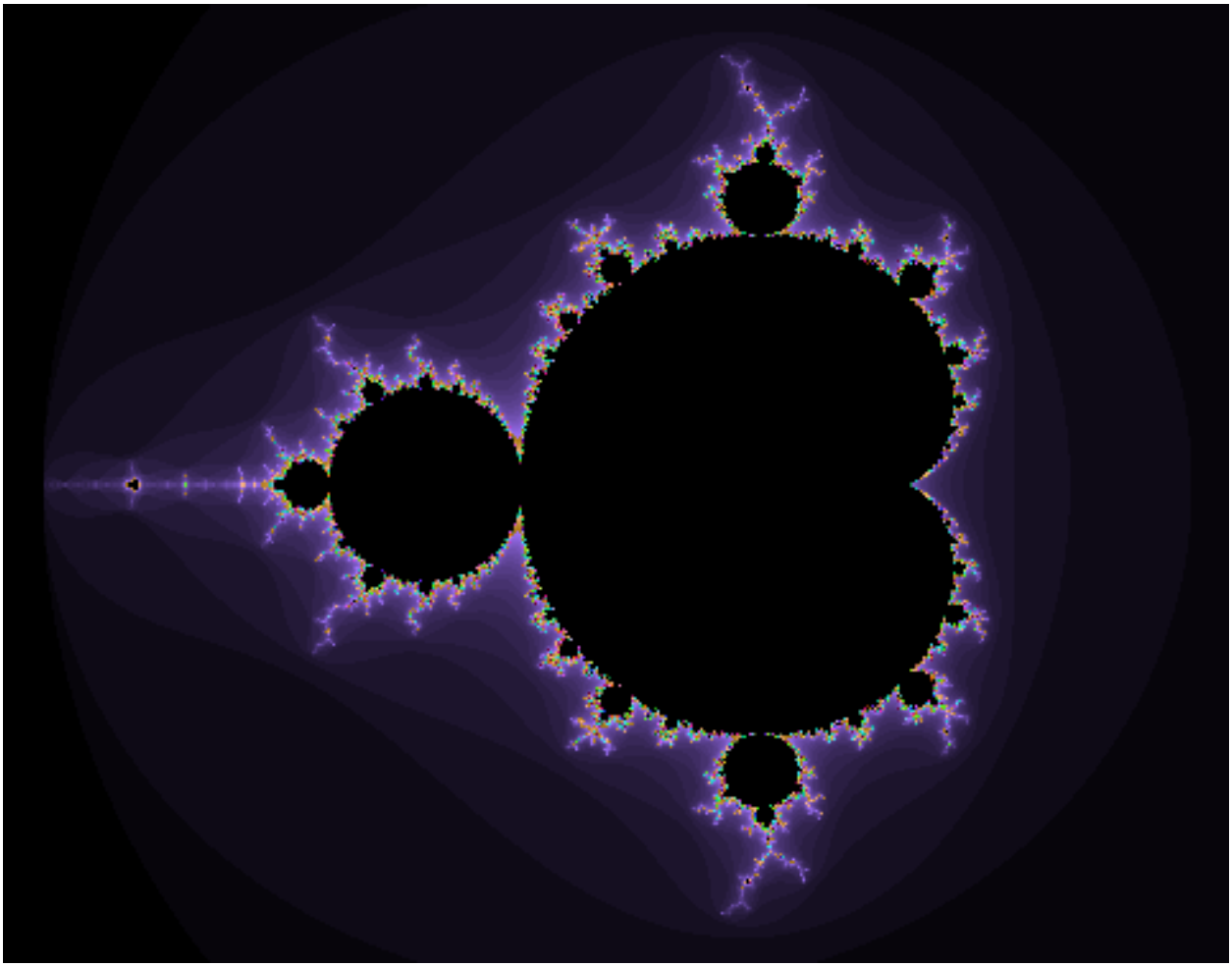
```
'''A conversion of Jon Skeet's LINQ Mandelbrot from LINQ to asq.

The original can be found at

http://msmvps.com/blogs/jon_skeet/archive/2008/02/26/visualising-the-mandelbrot-set-
↪with-linq-yet-again.aspx

'''
import colorsys
#import Image

from asq.initiators import integers, query
```

```
def generate(start, func):
    value = start
    while True:
        yield value
        value = func(value)


def colnorm(r, g, b):
    return (int(255 * r) - 1, int(255 * g) - 1, int(255 * b) - 1)


def col(n, max):
    if n == max:
        return (0, 0, 0)
    return colnorm(colorsys.hsv_to_rgb(0.0, 1.0, float(n) / max))


def mandelbrot():
    MaxIterations = 200
    SampleWidth = 3.2
    SampleHeight = 2.5
    OffsetX = -2.1
    OffsetY = -1.25

    ImageWidth = 480
    ImageHeight = int(SampleHeight * ImageWidth / SampleWidth)

    query = integers(0, ImageHeight).select(lambda y: (y * SampleHeight) /␣
↪ImageHeight + OffsetY) \
            .select_many_with_correspondence(
                lambda y: integers(0, ImageWidth).select(lambda x: (x * SampleWidth) /
↪ ImageWidth + OffsetX),
                lambda y, x: (x, y)) \
            .select(lambda real_imag: complex(*real_imag)) \
            .select(lambda c: query(generate(c, lambda x: x * x + c))
                              .take_while(lambda x: x.real ** 2 + x.imag ** 2 < 4)
                              .take(MaxIterations)
                              .count()) \
            .select(lambda c: ((c * 7) % 255, (c * 5) % 255, (c * 11) % 255) if c !=␣
↪MaxIterations else (0, 0, 0))

    data = q.to_list()

    image = Image.new("RGB", (ImageWidth, ImageHeight))
    image.putdata(data)
    image.show()


if __name__ == '__main__':
    mandelbrot()
```

This example can be be run with:

```
python -m asq.examples.mandelbrot
```

which produces



CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

C

D

E