Squint Documentation

Release 0.2.0.dev0

Shawn Brown

Mar 29, 2020

Documentation

Version 0.2.0.dev0

Squint is a simple query interface for tabular data that's light-weight and easy to learn. A core feature of Squint is that **the structure of a query's selection determines the structure of its result**. With it you can:

- Select data using Python literals—sets, lists, dictionaries, etc.—and get results in the same format.
- Aggregate, map, filter, reduce, and otherwise manipulate data.
- Lazily iterate over results, write them to a file, or eagerly evaluate them in memory.
- Analyze data from CSV, Excel, SQL, and other data sources.

CHAPTER 1

Documentation

1.1 Tutorials

These tutorials are written with the intent that you follow along and type the examples into Python's interactive prompt yourself. This will give you hands-on experience working with Select, Query, and Result objects.

1.1.1 Making Selections

The following examples demonstrate squint's `Select` class. For these examples, we will use the following data set:

A	B	C
x	foo	20
x	foo	30
y	foo	10
y	bar	20
z	bar	10
z	bar	10

Get Started

Download the data set as a CSV file:

`example.csv`

Start the Interactive Prompt

Open a command prompt and navigate to the folder that contains the example data. Then start Python in interactive mode so you can type commands at the >>> prompt:

```
$ python3
Python 3.8.0 (default, Oct 16 2019, 12:47:36)
[GCC 9.2.1 20190827 (Red Hat 9.2.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Supported Formats

Using *Select*, you can load data from difference sources:

- CSV files
- Database connections
- MS Excel files
- DBF files
- Pandas objects: DataFrame, Series, Index, or MultiIndex

You can also use shell-style wildcards to load multiple files into a single Select object:

```
select = Select('*.csv')
```

Load the Data

Import Squint and load the CSV data into a *Select* object:

```
>>> import squint
>>> select = squint.Select('example.csv')
```

Inspect Field Names

The *fieldnames* attribute contains a list of field names used in the data:

```
>>> select.fieldnames
['A', 'B', 'C']
```

Select Elements

A Select object can be called like a function—doing so returns a *Query* object.

Select a list of elements from column **A**:

```
>>> select('A')
Query(<squint.Select object at 0x7f02919d>, ['A'])
---- preview ----
['x', 'x', 'y', 'y', 'z', 'z']
```

Above, look at the "preview" and notice that these values come from column **A** in our *data set*.

Select a list of `tuple` elements from columns **A** and **B**, `('A', 'B')`:

```
>>> select(('A', 'B'))
Query(<squint.Select object at 0x7f02919d>, [('A', 'B')])
---- preview ----
[('x', 'foo'), ('x', 'foo'), ('y', 'foo'), ('y', 'bar'),
 ('z', 'bar'), ('z', 'bar')]
```

Select a list of `list` elements from columns **A** and **B**, `['A', 'B']`:

```
>>> select(['A', 'B'])
Query(<squint.Select object at 0x7f02919d>, [['A', 'B']])
---- preview ----
[['x', 'foo'], ['x', 'foo'], ['y', 'foo'], ['y', 'bar'],
 ['z', 'bar'], ['z', 'bar']]
```

The container type used in a selection determines the container types returned in the result. You can think of the selection as a template that describes the values and data types that are returned.

Note: In the examples above, we did not specify an outer-container type and—when unspecified—a `list` is used. So the outer-containers for all of the previous results were lists: a list of strings, a list of tuples, and a list of lists.

Specify Outer-Container Data Types

Compatible sequence and set types can be selected as inner- and outer-containers as needed. To specify an outer-container type, provide one of the following:

- a container that holds a single field name
- a container that holds another container (this second, inner-container can hold one or more field names)

Select a `set` of elements from column **A**, `{'A'}`:

```
>>> select({'A'})
Query(<squint.Select object at 0x7f02919d>, {'A'})
---- preview ----
{'x', 'y', 'z'}
```

Select a `set` of `tuple` elements from columns **A** and **B**, `{('A', 'B')}`:

```
>>> select({('A', 'B')})
Query(<squint.Select object at 0x7f02919d>, {('A', 'B')})
---- preview ----
{('x', 'foo'), ('y', 'foo'), ('y', 'bar'), ('z', 'bar')}
```

Tip: As mentioned previously, the default outer-container is a list, so when an early example used `select('A')`, that was actually a shorthand for `select(['A'])`. Likewise, `select(('A', 'B'))`, was a shorthand for `select([('A', 'B')])`.

Select Groups of Elements

To select groups of elements, use a `dict` (or other mapping type) as the outer-container—this dictionary must hold a single key-value pair. The *key* elements determine the "groups" used to arrange the results. And *value* elements are assigned to the same group when their associated keys are the same.

Select groups arranged by elements from column **A** that contain lists of elements from column **B**, `{'A': 'B'}`:

```
>>> select({'A': 'B'})
Query(<squint.Select object at 0x7f02919d>, {'A': ['B']})
---- preview ----
{'x': ['foo', 'foo'], 'y': ['foo', 'bar'], 'z': ['bar', 'bar']}
```

Select groups arranged by elements from column **A** that contain lists of `tuple` elements from columns **B** and **C**, `{'A': ('B', 'C')}`:

```
>>> select({'A': ('B', 'C')})
Query(<squint.Select object at 0x7f8cbc77>, {'A': [('B', 'C')]})
---- preview ----
{'x': [('foo', '20'), ('foo', '30')],
 'y': [('foo', '10'), ('bar', '20')],
 'z': [('bar', '10'), ('bar', '10')]}
```

To group by multiple columns, we use a `tuple` of key fields. Select groups arranged by elements from columns **A** and **B** that contain lists of elements from column **C**, `{('A', 'B'): 'C'}`:

```
>>> select({('A', 'B'): 'C'})
Query(<squint.Select object at 0x7f8cbc77>, {('A', 'B'): ['C']})
---- preview ----
{('x', 'foo'): ['20', '30'],
 ('y', 'bar'): ['20'],
 ('y', 'foo'): ['10'],
 ('z', 'bar'): ['10', '10']}
```

Specify Container Types for Groups

When selecting groups of elements, you can specify inner- and outer-container types for the *value*. The previous groupings used the default `list` shorthand. But as with non-grouped selections, you can specify a type explicitly.

Select groups arranged by elements from column **A** that contain `sets` of elements from column **B**, `{'A': {'B'}}`:

```
>>> select({'A': {'B'}})
Query(<squint.Select object at 0x7f2c36ee>, {'A': {'B'}})
---- preview ----
{'x': {'foo'}, 'y': {'foo', 'bar'}, 'z': {'bar'}}
```

Select groups arranged by elements from column **A** that contain `sets` of `tuple` elements from columns **B** and **C**, `{'A': {('B', 'C')}}`:

```
>>> select({'A': {('B', 'C')}})
Query(<squint.Select object at 0x7fc4a060>, {'A': {('B', 'C')}})
---- preview ----
{'x': {('foo', '30'), ('foo', '20')},
 'y': {('foo', '10'), ('bar', '20')},
 'z': {('bar', '10')}}
```

Narrowing a Selection

Selections can be narrowed to rows that satisfy given keyword arguments.

Narrow a selection to rows where column **B** equals "foo", `B='foo'`:

```
>>> select(('A', 'B'), B='foo')
Query(<squint.Select object at 0x7f978939>, [('A', 'B')], B='foo')
---- preview ----
[('x', 'foo'), ('x', 'foo'), ('y', 'foo')]
```

The keyword column does not have to be in the selected result:

```
>>> select('A', B='foo')
Query(<squint.Select object at 0x7f978939>, ['A'], B='foo')
---- preview ----
['x', 'x', 'y']
```

Narrow by Multiple Columns

Narrow a selection to rows where column **A** equals "y" *and* column **B** equals "bar", `A='y', B='bar'`:

```
>>> select(('A', 'B', 'C'), A='y', B='bar')
Query(<squint.Select object at 0x7f97893>, [('A', 'B', 'C')], A='y', B='bar')
---- preview ----
[('y', 'bar', '20')]
```

Only one row matches the above keyword conditions.

Narrow by Other Predicates

The argument's *key* specifies the column to check and its *value* is used to construct a *Predicate* that checks for matching elements. In addition to matching values like `'y'` or `'bar'`, Predicate objects can be sets, functions, boolean values, and more.

Use a predicate `set` to narrow a selection to rows where column **A** equals "x" *or* "y", `A={'x', 'y'}`:

```
>>> select(('A', 'B'), A={'x', 'y'})
Query(<squint.Select object at 0x7f97893>, [('A', 'B')], A={'y', 'x'})
---- preview ----
[('x', 'foo'), ('x', 'foo'), ('y', 'foo'), ('y', 'bar')]
```

Use a predicate *function* to narrow a selection to rows where column **C** is greater than 15, `C=greaterthan15`:

```
>>> def greaterthan15(x):
...     return float(x) > 15
...
>>> select(('A', 'C'), C=greaterthan15)
Query(<squint.Select object at 0x7fa6b9ea>, [('A', 'C')], C=greaterthan15)
---- preview ----
[('x', '20'), ('x', '30'), ('y', '20')]
```

When functions are simple like the one above, you can use a *lambda* statement rather than writing a separate function, `C=lambda x:  float(x) > 15`:

```
>>> select(('A', 'C'), C=lambda x: float(x) > 15)
Query(<squint.Select object at 0x7f5f08e4>, [('A', 'C')], C=<lambda>)
---- preview ----
[('x', '20'), ('x', '30'), ('y', '20')]
```

In addition to set membership and function testing, Predicates can be used for type checking, regex matching, and more. See the *Predicate documentation* for details.

Getting the Data Out

The examples so far have called `Select` objects and gotten `Query` objects in return. While the preview shows what the output will look like, it's still a Query object—not the data itself. One way to get the actual data is to use the Query's `fetch()` method.

Get the data out by calling the `fetch()` method:

```
>>> select('A').fetch()
['x', 'x', 'y', 'y', 'z', 'z']
```

1.1.2 Building Queries

The following examples demonstrate squint's `Query` class. This document builds on the *Making Selections* tutorial.

Get Started

We will get started the same way we did in the first tutorial. Begin by starting the Python interactive prompt in the same directory as the `example.csv` file. Once you are at the >>> prompt, import squint and load the data:

```
>>> import squint
>>> select = squint.Select('example.csv')
```

Creating a Query Object

In the *Making Selections* tutorial, we created several Query objects—each call to a `Select` object returns a Query.

By selecting a list of elements from column **C**, we get a `Query` object in return:

```
>>> select('C')
Query(<squint.Select object at 0x7ffa625b>, ['C'])
---- preview ----
['20', '30', '10', '20', '10', '10']
```

We can also create Queries directly using the following syntax (although it's rarely necessary to do so):

```
>>> squint.Query(select, 'C')
Query(<squint.Select object at 0x7ffa625b>, ['C'])
---- preview ----
['20', '30', '10', '20', '10', '10']
```

Once a Query has been created, we can perform additional operations on it using the methods described below.

Aggregate Methods

Aggregate methods operate on a collection of elements and produce a single result. The Query class provides several aggregate methods: `sum()`, `avg()`, `min()`, `max()`, and `count()`. For more information see the *aggregate methods* reference documentation.

Use the `sum()` method to sum the elements in column **C**:

```
>>> select('C').sum()
Query(<squint.Select object at 0x7ffa625b>, ['C']).sum()
---- preview ----
100
```

When an aggregate method is called on a `dict` or other mapping, the groups—the dictionary values—are operated on separately.

Use the *`sum()`* method to sum each group of elements:

```
>>> select({'A': 'C'}).sum()
Query(<squint.Select object at 0x7ffa625b>, {'A': ['C']}).sum()
---- preview ----
{'x': 50, 'y': 30, 'z': 20}
```

Type Conversion

The *`Query`* class contains two methods that perform automatic type conversion:

- *`sum()`*
- *`avg()`*

In the example above, column **C** contains `str` elements. These strings are automatically converted to `float` values. The other functional methods do not do this—use *`map()`* to convert values explicitly.

Functional Methods

Functional methods take a user-provided function and use it to perform a specified procedure. The Query class provides the following functional methods: *`map()`*, *`filter()`*, *`reduce()`*, *`apply()`*, etc. For more information see the *functional methods* reference documentation.

Use the *`map()`* method to apply a function to each element:

```
>>> def uppercase(value):
...     return value.upper()
...
>>> select('B').map(uppercase)
Query(<squint.Select object at 0x7ffa625b>, ['B']).map(uppercase)
---- preview ----
['FOO', 'FOO', 'FOO', 'BAR', 'BAR', 'BAR']
```

Use the *`filter()`* method to narrow the selection to items for which the function returns True:

```
>>> def not_bar(value):
...     return value != 'bar'
...
>>> select('B').filter(not_bar)
Query(<squint.Select object at 0x7ffa625b>, ['B']).filter(not_bar)
---- preview ----
['foo', 'foo', 'foo']
```

Element-Wise vs Group-Wise Methods

The *`map()`*, *`filter()`*, and *`reduce()`* methods perform element-wise procedures—they call their user-provided functions for each element and do something with the result. The *`apply()`* method, however, performs a group-wise

procedure. Rather than calling its user-provided function for each element, it calls the function once per *container* of elements.

Use the *apply()* method to apply a function to an entire container of elements:

```
>>> def join_strings(container):
...     return '-'.join(container)
...
>>> select('B').apply(join_strings)
Query(<squint.Select object at 0x7ffa625b>, ['B']).apply(join_strings)
---- preview ----
'foo-foo-foo-bar-bar-bar'
```

Like the aggregate methods, when *apply()* is called on a `dict` or other mapping, the groups—the dictionary values—are operated on separately.

Use the *apply()* method to apply a function for each container of elements:

```
>>> select({'A': 'B'}).apply(join_strings)
Query(<squint.Select object at 0x7ffa625b>, {'A': ['B']}).apply(join_strings)
---- preview ----
{'x': 'foo-foo', 'y': 'foo-bar', 'z': 'bar-bar'}
```

Data Handling Methods

Data handling methods operate on a collection of elements by reshaping or otherwise reformatting the data. The Query class provides the following data handling methods: *flatten()*, *unwrap()*, and *distinct()*. For more information see the *data handling methods* reference documentation.

The *flatten()* method serializes a `dict` or other mapping into list of tuple rows. Let's start by observing the structure of a selected dictionary `{'B': 'C'}`:

```
>>> select({'B': 'C'})
Query(<squint.Select object at 0x7ffa625b>, {'B': ['C']})
---- preview ----
{'foo': ['20', '30', '10'],
 'bar': ['20', '10', '10']}
```

Now, use the *flatten()* method to serialize this same selection (`{'B': 'C'}`) into a list of tuples:

```
>>> select({'B': 'C'}).flatten()
Query(<squint.Select object at 0x7ffa625b>, {'B': ['C']}).flatten()
---- preview ----
[('foo', '20'), ('foo', '30'), ('foo', '10'),
 ('bar', '20'), ('bar', '10'), ('bar', '10')]
```

The *unwrap()* method unwraps single-element containers and returns the element itself. Multi-element containers are untouched. Observe the structure of the following preview, `{('A', 'B'): 'C'}`:

```
>>> select({('A', 'B'): 'C'})
Query(<squint.Select object at 0x7ffa625b>, {('A', 'B'): ['C']})
---- preview ----
{('x', 'foo'): ['20', '30'],
 ('y', 'bar'): ['20'],
 ('y', 'foo'): ['10'],
 ('z', 'bar'): ['10', '10']}
```

Use the *unwrap()* method to unwrap `['20']` and `['10']` but leave the multi-element lists untouched:

```
>>> select({('A', 'B'): 'C'}).unwrap()
Query(<squint.Select object at 0x7ffa625b>, {('A', 'B'): ['C']}).unwrap()
---- preview ----
{('x', 'foo'): ['20', '30'],
 ('y', 'bar'): '20',
 ('y', 'foo'): '10',
 ('z', 'bar'): ['10', '10']}
```

Data Output Methods

Data output methods evaluate the query and return its results. The Query class provides the following data output methods: *fetch()*, *execute()* and *to_csv()*. For more information see the *data output methods* reference documentation.

Use the *fetch()* method to eagerly evaluate the query and return its results:

```
>>> select('A').fetch()
['x', 'x', 'y', 'y', 'z', 'z']
```

Use the *execute()* method to lazily evaluate the query by returning a *Result* object:

```
>>> select('A').execute()
<Result object (evaltype=list) at 0x7fa32d16>
```

Eager vs Lazy Evaluation

When a query is *eagerly evaluated*, its elements are all loaded into memory at the same time. But when a query is *lazily evaluated*, its individual elements are computed one-at-a-time. See the *Using Results* tutorial for more information about *eager* and *lazy* evaluation.

Use the *to_csv()* method to save the query results into a CSV file:

```
>>> select('A').to_csv('myresults.csv')
```

Method Chaining

You can build increasingly complex queries by chaining methods together as needed:

```
>>> def not_z(value):
...     return value != 'z'
...
>>> def uppercase(value):
...     return str(value).upper()
...
>>> select('A').filter(not_z).map(uppercase).fetch()
['X', 'X', 'Y', 'Y']
```

In the example above, the *filter()*, *map()*, and *fetch()* methods are chained together to perform multiple operations within a single statement and then output the data.

Method Order

The order of most Query methods can be mixed and matched as needed. But the data output methods—like *fetch()*, *execute()*, and *to_csv()*—can only appear at the end of a chain, not in the middle of one.

1.1.3 Using Results

The following examples demonstrate squint's *Result* class. This document builds on the previous *Making Selections* and *Building Queries* tutorials.

Get Started

We will get started the same way we did in the previous tutorials. Begin by starting the Python interactive prompt in the same directory as the `example.csv` file. Once you are at the >>> prompt, import squint and load the data:

```
>>> import squint
>>> select = squint.Select('example.csv')
```

Creating a Result Object

Typically, we create *Result* objects by calling a Query's *execute()* method:

```
>>> select('A').execute()
<Result object (evaltype=list) at 0x7ff5f372>
```

We can also create Results directly with the following syntax:

```
>>> iterable = [1, 2, 3, 4, 5]
>>> squint.Result(iterable, evaltype=list)
<Result object (evaltype=list) at 0x7ff5f38d>
```

The `evaltype` Attribute

The *evaltype* attribute—short for "evaluation type"—indicates the type of container that a Result represents:

```
>>> result = select('A').execute()
>>> result.evaltype
<class 'list'>
```

Eager Evaluation

When a Result is *eagerly evaluated*, all of its contents are loaded into memory at the same time. Doing this returns an container of elements whose type is determined by the Result's *evaltype*.

Use the *fetch()* method to eagerly evaluate the result and get its contents:

```
>>> result = select('A').execute()
>>> result.fetch()
['x', 'x', 'y', 'y', 'z', 'z']
```

For many results, eager evaluation is entirely acceptible. But large results might use a lot of memory or even exceed the memory available on your system.

Lazy Evaluation

When a Result is *lazily evaluated*, its individual elements are computed one-at-a-time as they are needed. In fact, the primary purpose of a Result object is to facilitate lazy evaluation when possible.

Use a `for` loop to lazily evaluate the result and get its contents:

```
>>> result = select('A').execute()
>>> for element in result:
...     print(element)
...
...
x
x
y
y
z
z
```

For each iteration of the loop in the above example, the next element is evaluated and the previous element is discarded. At no point in time do all of the elements occupy memory together.

Note: When lazily evaluating a Result, you are free to check the *`evaltype`* but it is never actually used to create an object of that type.

1.2 How-To Guide

Many of the following sections use the example CSV from the tutorials. You can download it here:

`example.csv`

1.2.1 How To Install Squint

The Squint package is tested on Python 2.7, 3.4 through 3.8, PyPy, and PyPy3; and is freely available under the Apache License, version 2.

The easiest way to install squint is to use pip:

```
pip install squint
```

To upgrade an existing installation, use the "`--upgrade`" option:

```
pip install --upgrade squint
```

The development repository for `squint` is hosted on GitHub. If you need bug-fixes or features that are not available in the current stable release, you can "pip install" the development version directly from GitHub:

```
pip install --upgrade https://github.com/shawnbrown/squint/archive/master.zip
```

All of the usual caveats for a development install should apply—only use this version if you can risk some instability or if you know exactly what you're doing. While care is taken to never break the build, it can happen.

1.2.2 How To Convert an Element's Type

To change the data type of individual elements, use the *map()* method to apply a type to each element.

In the following example, we convert string elements into the float type, `map(float)`:

```
>>> import squint
>>>
>>> select = squint.Select('example.csv')
>>>
>>> select('C').map(float)
Query(<squint.Select object at 0x7fcaac15>, ['C']).map(float)
---- preview ----
[20.0, 30.0, 10.0, 20.0, 10.0, 10.0]
```

In the preview above, we see that every element in column **C** has been converted into a `float` value.

1.2.3 How To Convert a Container's Type

While you can control a container's type *during* selection, there are times when you will want to convert a container's type *after* selection. To do this, use the *apply()* method to apply a container type to the entire group of elements.

In the following example, we convert a list of elements into a tuple of elements, `apply(tuple)`:

```
>>> import squint
>>>
>>> select = squint.Select('example.csv')
>>>
>>> select('A').apply(tuple)
Query(<squint.Select object at 0x7f8ed8b6>, ['A']).apply(tuple)
---- preview ----
('x', 'x', 'y', 'y', 'z', 'z')
```

In the preview above, we see that our query returns a `tuple` instead of a list.

1.2.4 How To Select Single-Item Inner-Containers

To specify a single-item inner-container, you must provide both inner- and outer-types explicitly.

For example, select single-item `sets` of elements from column **B**, `[{'B'}]`:

```
>>> import squint
>>>
>>> select = squint.Select('example.csv')
>>>
>>> select([{'B'}])
Query(<squint.Select object at 0x7ff9292f>, [{'B'}])
---- preview ----
[{'foo'}, {'foo'}, {'foo'}, {'bar'}, {'bar'}, {'bar'}]
```

This is necessary because a single-item container—when used by itself—specifies an outer-container type. You cannot use the implicit `list` shorthand demonstrated elsewhere in the documentation.

1.2.5 How To Select Exotic Data Types

Most examples demonstrate the use of squint's *Select* class with list, tuple and set types, but it's possible to use a wide variety of other containers, too. For instance, frozensets, deques, namedtuples, etc. can be used the same way you would use any of the previously mentioned types.

For example, select a deque of namedtuple elements from columns **A** and **B**, deque([ntup('A', 'B')]):

```
>>> from collections import deque
>>> from collections import namedtuple
>>> import squint
>>>
>>> select = squint.Select('example.csv')
>>>
>>> ntup = namedtuple('ntup', ['first', 'second'])
>>>
>>> select(deque([ntup('A', 'B')]))
Query(<squint.Select object at 0x7f4cf01c>, deque([ntup(first='A', second='B')]))
---- preview ----
deque([ntup(first='x', second='foo'), ntup(first='x', second='foo'),
       ntup(first='y', second='foo'), ntup(first='y', second='bar'),
       ntup(first='z', second='bar'), ntup(first='z', second='bar')])
```

Note: You can mix and match container types as desired, but the normal object limitations still apply. For example, sets and dictionary keys can only contain immutable types (like str, tuple, frozenset, etc.).

1.3 API Reference

1.3.1 Select

class squint.**Select**(*objs=None*, **args*, ***kwds*)

A class to quickly load and select tabular data. The given *objs*, **args*, and ***kwds*, can be any values supported by get_reader(). Additionally, *objs* can be a list of supported objects or a string with shell-style wildcards. If *objs* is already a reader-like object, it will be used as is.

Load a single file:

```
select = datatest.Select('myfile.csv')
```

Load a reader-like iterable:

```
select = datatest.Select([
    ['A', 'B'],
    ['x', 100],
    ['y', 200],
    ['z', 300],
])
```

Load multiple files:

```
select = datatest.Select(['myfile1.csv', 'myfile2.csv'])
```

Load multple files using a shell-style wildcard:

```
select = datatest.Select('*.csv')
```

When multiple sources are loaded into a single Select, data is aligned by fieldname and missing fields receive empty strings:

load_data(*objs*, **args*, ***kwds*)

Load data from one or more objects into the Select. The given *objs*, **args*, and ***kwds*, can be any values supported by the *Select* class initialization.

Load a single file into an empty Select:

```
select = datatest.Select()  # <- Empty Select.
select.load_data('myfile.csv')
```

Add a single file to an already-populated Select:

```
select = datatest.Select('myfile1.csv')
select.load_data('myfile2.xlsx', worksheet='Sheet2')
```

Add multiple files to an already-populated Select:

```
select = datatest.Select('myfile1.csv')
select.load_data(['myfile2.csv', 'myfile3.csv'])
```

fieldnames

A list of field names used by the data source.

__call__(*columns=None*, ***where*)

After a Select has been created, it can be called like a function to select fields and return an associated *Query* object.

The *columns* argument serves as a template to define the values and data types selected. All *columns* selections will be wrapped in an outer container. When a container is unspecified, a `list` is used as the default:

```
select = datatest.Select('example.csv')
query = select('A')  # <- selects a list of values from 'A'
```

When *columns* specifies an outer container, it must hold only one field—if a given container holds multiple fields, it is assumed to be an inner container (which gets wrapped in the default outer container):

```
query = select(('A', 'B'))  # <- selects a list of tuple
                            #    values from 'A' and 'B'
```

When *columns* is a `dict`, values are grouped by key:

```
query = select({'A': 'B'})  # <- selects a dict with
                            #    keys from 'A' and
                            #    values from 'B'
```

When *columns* is omitted, the object's *fieldnames* are used instead.

Optional *where* keywords can narrow the selected data to matching rows. A key must specify the field to check and a value must be a predicate object (see *Predicate* for details). Rows where the predicate is a match are selected and rows where it doesn't match are excluded:

```
select = datatest.Select('example.csv')
query = select({'A'}, B='foo')  # <- selects only the rows
                                #    where 'B' equals 'foo'
```

See the *Making Selections* tutorial for step-by-step examples.

create_index(**columns*)

Create an index for specified columns—can speed up testing in many cases.

If you repeatedly use the same few columns to group or filter results, then you can often improve performance by adding an index for these columns:

```
select.create_index('town')
```

Using two or more columns creates a multi-column index:

```
select.create_index('town', 'postal_code')
```

Calling the function multiple times will create multiple indexes:

```
select.create_index('town')
select.create_index('postal_code')
```

Note: Indexes should be added with discretion to tune a test suite's over-all performance. Creating several indexes before testing even begins could lead to longer run times so use indexes with care.

1.3.2 Query

class squint.**Query**(*columns*, ***where*)

class squint.**Query**(*select*, *columns*, ***where*)

A class to query data from a source object. Queries can be created, modified, and passed around without actually computing the result—computation doesn't occur until the query object itself or its *fetch()* method is called.

The given *columns* and *where* arguments can be any values supported by *Select.__call__()*.

Although Query objects are usually created by `calling` an existing Select, it's possible to create them independent of any single data source:

```
query = Query('A')
```

classmethod from_object(*obj*)

Creates a query and associates it with the given object.

```
mylist = [1, 2, 3, 4]
query = Query.from_object(mylist)
```

If *obj* is a Query itself, a copy of the original query is created.

AGGREGATE METHODS

Aggregate methods operate on a collection of elements and produce a single result.

sum()

Get the sum of non-None elements.

avg()
Get the average of non-None elements. Strings and other objects that do not look like numbers are interpreted as 0.

min()
Get the minimum value from elements.

max()
Get the maximum value from elements.

count()
Get the count of non-None elements.

FUNCTIONAL METHODS

Functional methods take a user-provided function and use it to perform a specified procedure.

apply(*function*)
Apply *function* to entire group keeping the resulting data. If element is not iterable, it will be wrapped as a single-item list.

map(*function*)
Apply *function* to each element, keeping the results. If the group of data is a set type, it will be converted to a list (as the results may not be distinct or hashable).

filter(*predicate=True*)
Filter elements, keeping only those values that match the given *predicate*. When *predicate* is True, this method keeps all elements for which `bool` returns True (see *Predicate* for details).

reduce(*function*, *initializer_factory=None*)
Reduce elements to a single value by applying a *function* of two arguments cumulatively to all elements from left to right. If the optional *initializer_factory* is present, it is called without arguments to provide a value that is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If initializer_factory is not given and sequence contains only one item, the first item is returned.

starmap(*function*)

DATA HANDLING METHODS

Data handling methods operate on a collection of elements by reshaping or otherwise reformatting the data.

distinct()
Filter elements, removing duplicate values.

flatten()
Flatten dictionary into list of tuple rows. If data is not a dictionary, the original values are returned unchanged.

unwrap()
Unwrap single-item sequences or sets.

DATA OUTPUT METHODS

Data output methods evaluate the query and return its results.

execute(*source=None*, *optimize=True*)
A Query can be executed to return a single value or an iterable *Result* appropriate for lazy evaluation:

```
query = source('A')
result = query.execute()  # <- Returns Result (iterator)
```

Setting *optimize* to False turns-off query optimization.

fetch()
Executes query and returns an eagerly evaluated result.

to_csv(*file*, *fieldnames=None*, ***fmtparams*)
Execute the query and write the results as a CSV file (dictionaries and other mappings will be seralized).

The given *file* can be a path or file-like object; *fieldnames* will be printed as a header row; and *fmtparams* can be any values supported by `csv.writer()`.

When *fieldnames* are not provided, names from the query's original *columns* argument will be used if the number of selected columns matches the number of resulting columns.

1.3.3 Result

class squint.**Result**(*iterable*, *evaltype*, *closefunc=None*)
A simple iterator that wraps the results of *Query* execution to facilitate lazy evaluation of the resulting data.

Although Result objects are usually constructed automatically, it's possible to create them directly:

```
iterable = iter([...])
result = Result(iterable, evaltype=list)
```

Warning: When iterated over, the *iterable* **must** yield only those values necessary for constructing an object of the given *evaltype* and no more. For example, when the *evaltype* is a set, the *iterable* must not contain duplicate or unhashable values. When the *evaltype* is a `dict` or other mapping, the *iterable* must contain unique key-value pairs or a mapping.

evaltype
The type of instance returned by the *fetch* method.

fetch()
Evaluate the entire iterator and return its result:

```
result = Result(iter([...]), evaltype=set)
result_set = result.fetch()  # <- Returns a set of values.
```

When evaluating a `dict` or other mapping type, any values that are, themselves, *Result* objects will also be evaluated.

__wrapped__
The underlying iterator—useful when introspecting or rewrapping.

1.3.4 Predicate

Squint can use *Predicate* objects for narrowing and filtering selections.

class squint.**Predicate**(*obj*, *name=None*)
A Predicate is used like a function of one argument that returns `True` when applied to a matching value and `False` when applied to a non-matching value. The criteria for matching is determined by the *obj* type used to define the predicate:

obj type	matches when
function	the result of `function(value)` tests as True
type	value is an instance of the type
`re.compile(pattern)`	value matches the regular expression pattern
`True`	value is truthy (`bool(value)` returns True)
`False`	value is falsy (`bool(value)` returns False)
str or non-container	value is equal to the object
set	value is a member of the set
tuple of predicates	tuple of values satisfies corresponding tuple of predicates—each according to their type
`...` (Ellipsis literal)	(used as a wildcard, matches any value)

Example matches:

obj example	value	matches
`def iseven(x):` `  return x % 2 == 0`	`4`	Yes
	`9`	No
`float`	`1.0`	Yes
	`1`	No
`re.compile('[bc]ake')`	`'bake'`	Yes
	`'cake'`	Yes
	`'fake'`	No
`True`	`'x'`	Yes
	`''`	No
`False`	`''`	Yes
	`'x'`	No
`'foo'`	`'foo'`	Yes
	`'bar'`	No
`{'A', 'B'}`	`'A'`	Yes
	`'C'`	No
`('A', float)`	`('A', 1.0)`	Yes
	`('A', 2)`	No
`('A', ...)` Uses ellipsis wildcard.	`('A', 'X')`	Yes
	`('A', 'Y')`	Yes
	`('B', 'X')`	No

Example code:

```
>>> pred = Predicate({'A', 'B'})
>>> pred('A')
True
```

(continues on next page)

(continued from previous page)

```
>>> pred('C')
False
```

Predicate matching behavior can also be inverted with the inversion operator (~). Inverted Predicates return `False` when applied to a matching value and `True` when applied to a non-matching value:

```
>>> pred = ~Predicate({'A', 'B'})
>>> pred('A')
False
>>> pred('C')
True
```

If the *name* argument is given, a `__name__` attribute is defined using the given value:

```
>>> pred = Predicate({'A', 'B'}, name='a_or_b')
>>> pred.__name__
'a_or_b'
```

If the *name* argument is omitted, the object will not have a `__name__` attribute:

```
>>> pred = Predicate({'A', 'B'})
>>> pred.__name__
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    pred.__name__
AttributeError: 'Predicate' object has no attribute '__name__'
```

CHAPTER 2

Other Resources

- Squint on PyPI
- Squint on GitHub

Python Module Index

S

Index

Symbols

A

C

D

E

F

L

M

P

Q

R

S

T

U