Import Tools Documentation

Release 0.1.1

Sever Banesiu

January 31, 2013

CONTENTS

This library is intended to be a set of reusable tools and concepts for writing simple to complex import jobs.

CHAPTER

ONE

TUTORIAL

To see an example of how this library can be used please read the *Import Tutorial*.

1.1 Import Tutorial

1.1.1 Subtitle1

1.1.2 Subtitle2

1.1.3 Subtitle3

CHAPTER

TWO

API REFERENCE

2.1 Importable Elements

This module defines some handy `Importable` elements.

An `Importable` is usually composed of two different parts:

- A *natural key* used to identify *the same* element across different systems. This is the only required component for an `Importable`.
- An optional set of properties that form *the contents*. The data in this properties is carried across systems in the process of syncing the elements.

Two elements that are *the same* and have *equal contents* are said to be *in sync*.

For example an element representing an online video can use the value of the streaming URL to be its natural key. The contents of the element can be formed from a view counter and the video title. In this scenario changes on the video title and view counter can be detected and carried across systems thus keeping elements which are the same in sync. Changes to the video URL will make the video element lose any correspondence with elements belonging to other systems.

class `importtools.importables.`**`Importable`**(*natural_key*, **args*, ***kwargs*)

A default implementation representing an importable element.

This class is intended to be specialized in order to provide the element content and to override its behaviour if needed.

The `sync()` implementation in this class doesn't keep track of changed values. For such an implementation see `RecordingImportable`.

`Importable` instances are hashable and comparable based on the *natural_key* value. Because of this the *natural_key* must also be hashable and should implement equality and less then operators:

```
>>> i1 = Importable(0)
>>> i2 = Importable(0)
>>> hash(i1) == hash(i2)
True
>>> i1 == i2
True
>>> not i1 < i2
True
```

`Importable` elements can access the *natural_key* value used on instantiation trough the `natural_key` property:

```
>>> i = Importable((123, 'abc'))
>>> i.natural_key
(123, 'abc')
```

Listeners can register to observe an `Importable` element for changes. Every time the content attributes change with a value that is not equal to the previous one all registered listeners will be notified:

```
>>> class MockImportable(Importable):
...     _content_attrs = ['a', 'b']
>>> i = MockImportable(0)

>>> notifications = []
>>> i.register(lambda x: notifications.append(x))
>>> i.a = []
>>> i.b = 'b'
>>> i.b = 'bb'
>>> len(notifications)
3
>>> notifications[0] is notifications[1] is notifications[2] is i
True

>>> notifications = []
>>> l = []
>>> i.a = l
>>> len(notifications)
0
>>> i.a is l
True
```

There is also a shortcut for defining new `Importable` classes other than using inheritance by setting `__content_attrs__` to an iterable of attribute names. This will automatically create a constructor for your class that accepts all values in the list as keyword arguments. It also sets `_content_attrs` and `__slots__` to include this values and generates a `__repr__` for you. This method however may not fit all your needs, in that case subclassing `Importable` is still your best option.

One thing to keep in mind is that it's not possible to dinamicaly change `_content_attrs` for instances created from this class because of the `__slots__` usage.

```
>>> class MockImportable(Importable):
...     __content_attrs__ = ['a', 'b']

>>> MockImportable(0)
MockImportable(0)

>>> MockImportable(0, a=1, b=('a', 'b'))
MockImportable(0, a=1, b=('a', 'b'))

>>> i = MockImportable(0, a=1)
>>> i.b = 2
>>> i.a, i.b
(1, 2)
>>> i.update(a=100, b=200)
True
```

update(***kwargs*)

Update multiple content attrtibutes and fire a single notification.

Multiple changes to the element content can be grouped in a single call to `update()`. This method should return `True` if at least one element differed from the original values or else `False`.

```
>>> class MockImportable(Importable):
...     _content_attrs = ['a', 'b']
>>> i = MockImportable(0)
>>> i.register(lambda x: notifications.append(x))

>>> notifications = []
>>> i.update(a=100, b=200)
True
>>> len(notifications)
1
>>> notifications[0] is i
True
>>> notifications = []
>>> i.update(a=100, b=200)
False
>>> len(notifications)
0
```

Trying to call update using keywords that are not present in `_content_attrs` souhld raise `ValueError`:

```
>>> i.update(c=1)
Traceback (most recent call last):
ValueError:
```

sync(*other*)

Puts this element in sync with the *other*.

The default implementation uses `_content_attrs` to search for the attributes that need to be synced between the elements and it copies the values of each attribute it finds from the *other* element in this one.

By default the `self._content_attrs` is an empty list so no synchronization will take place:

```
>>> class MockImportable(Importable):
...     pass
>>> i1 = MockImportable(0)
>>> i2 = MockImportable(0)

>>> i1.a, i1.b = 'a1', 'b1'
>>> i2.a, i2.b = 'a2', 'b2'

>>> has_changed = i1.sync(i2)
>>> i1.a
'a1'

>>> class MockImportable(Importable):
...     _content_attrs = ['a', 'b', 'x']
>>> i1 = MockImportable(0)
>>> i2 = MockImportable(0)
>>> i1.a, i1.b = 'a1', 'b1'
>>> i2.a, i2.b = 'a2', 'b2'

>>> has_changed = i1.sync(i2)
>>> i1.a, i1.b
('a2', 'b2')
```

If no synchronization was needed (i.e. the content of the elements were equal) this method should return `False`, otherwise it should return `True`:

```
>>> i1.sync(i2)
False
>>> i1.a = 'a1'
>>> i1.sync(i2)
True
```

If the sync mutated this element all listeners should be notified. See `register()`:

```
>>> i1.a = 'a1'
>>> notifications = []
>>> i1.register(lambda x: notifications.append(x))
>>> has_changed = i1.sync(i2)
>>> len(notifications)
1
>>> notifications[0] is i1
True
```

All attributes that can't be found in the *other* element are skipped:

```
>>> i1._content_attrs = ['a', 'b', 'c']
>>> has_changed = i1.sync(i2)
>>> hasattr(i1, 'c')
False
```

register(*listener*)

Register a callable to be notified when `sync` changes data.

This method should raise an `ValueError` if *listener* is not a callable:

```
>>> i = Importable(0)
>>> i.register(1)
Traceback (most recent call last):
ValueError:
```

Same listener can register multiple times:

```
>>> notifications = []
>>> listener = lambda x: notifications.append(x)
>>> i.register(listener)
>>> i.register(listener)
>>> i._notify()
>>> notifications[0] is notifications[1] is i
True
```

is_registered(*listener*)

Check if the listener is already registered.

```
>>> i = Importable(0)
>>> a = lambda x: None
>>> i.is_registered(a)
False
>>> i.register(a)
>>> i.is_registered(a)
True
```

_notify()

Sends a notification to all listeners passing this element.

class `importtools.importables.`**RecordingImportable**(**args*, ***kwargs*)

Bases: `importtools.importables.Importable`

Very similar to `Importable` but tracks changes.

This class records the original values that the attributes had before any change introduced by attribute assignment or call to `update` and `sync`.

Just as in `Importable` case you can define new classes using `__content_attrs__` as a shortcut.

```
>>> class MockImportable(RecordingImportable):
...     __content_attrs__ = ['a', 'b']

>>> MockImportable(0)
MockImportable(0)

>>> MockImportable(0, a=1, b=('a', 'b'))
MockImportable(0, a=1, b=('a', 'b'))

>>> i = MockImportable(0, a=1)
>>> i.b = 2
>>> i.a, i.b
(1, 2)
>>> i.update(a=100, b=200)
True
>>> i.orig.a
1
```

orig

An object that can be used to access the elements original values.

The object has all the attributes that this element had when it was instantiated or last time when `reset()` was called.

```
>>> class MockImportable(RecordingImportable):
...   _content_attrs = ['a']
>>> i = MockImportable(0)

>>> hasattr(i.orig, 'a')
False
>>> i.a = 'a'
>>> i.reset()
>>> i.a
'a'
>>> i.orig.a
'a'
>>> i.a = 'aa'
>>> i.a
'aa'
>>> i.orig.a
'a'
>>> del i.a
>>> i.reset()
>>> hasattr(i.orig, 'a')
False
```

reset()

Create a snapshot of the current values.

```
>>> class MockImportable(RecordingImportable):
...   _content_attrs = ['a']
>>> i = MockImportable(0)
```

```
>>> hasattr(i.orig, 'a')
False
>>> i.a = 'a'
>>> i.reset()
>>> i.a = 'aa'
>>> i.orig.a
'a'
>>> i.reset()
>>> i.orig.a
'aa'
```

2.2 DataSet Containers

This module contains `DataSet` implementations used to hold Importables.

The `DataSet` concrete classes should be used when running import algorithms as the destination. Before starting the import it should contain the initial data found in the destination system and after running the import it will contain the newly synchronized data. From this point of view this structure serves as both input and output argument for the algorithm.

class `importtools.datasets.`**DataSet**
An `abc` that represents a mutable set of elements.

This class serves as documentation of the methods a `DataSet` should implement. For concrete implementations available in this module see `SimpleDataSet` and `RecordingDataSet`.

A *DataSet* is very similar with a normal *set* the difference being that you can `get()` an element. This is useful is because if the elements are `Importable` instances even if they are equal (the natural keys are the same) the contents may be different.

__iter__()
Iterate over all the content of this set.

get(*element*, *default=None*)
Return an equal element from the dataset or the default value.

add(*element*)
Add or replace the element in the dataset.

pop(*element*, *default=None*)
Remove and return an equal element from the dataset.

sync(*iterable*)
Add, remove and update this elements with those in the iterable.

class `importtools.datasets.`**SimpleDataSet**(*data_loader=None*, **args*, ***kwargs*)
Bases: `dict`, `importtools.datasets.DataSet`

A simple `dict`-based `DataSet` implementation.

At first, a newly created instance has no elements:

```
>>> from importtools import Importable
>>> i1, i2, i3 = Importable(0), Importable(0), Importable(1)
>>> sds = SimpleDataSet()
>>> list(sds)
[]
```

After creation, it can be populated and the elements in the dataset can be retrieved using other equal elements. Trying to `get` an inexistent item should return the default value or `None`:

```
>>> sds.add(i1)
>>> sds.get(i1) is i1
True
>>> sds.get(i2) is i1
True
>>> sds.get(i3) is None
True
>>> sds.get(i3, 'default')
'default'
>>> sds.pop(i3, 'default')
'default'
>>> sds.pop(i1)
Importable(0)
```

An iterable containing the initial data can be passed when constructing intances:

```
>>> SimpleDataSet((i1, i3))
SimpleDataSet([Importable(0), Importable(1)])
```

A *ValueError* should be raised if the initial data contains duplicates:

```
>>> init_values = (i1, i2, i3)
>>> SimpleDataSet(init_values)
Traceback (most recent call last):
ValueError:
```

class `importtools.datasets.`**`RecordingDataSet`**(*data_loader=()*, **args*, ***kwargs*)
Bases: `importtools.datasets.SimpleDataSet`

A `DataSet` implementation that remembers all the changes, additions and removals done to it.

Using instances of this calss as the destination of the import algorithm allows optimal persistence of the changes by grouping them in a way suited for batch processing.

`reset`()
Forget all recorded changes.

Calling this method will empty out *added*, *removed* and *changed*.

`added`
An iterable of all added elements in the dataset.

`removed`
An iterable of all removed elements in the dataset.

`changed`
An iterable of all elements that have been changed.

Only the elements that were part of the set from the beginning or before the last call to reset will be tracked. Deleting an element that has changed will not remove it from this list. This means it's possible for an element to be present in both `changed` and `removed` iterables.

- *genindex*
- *modindex*

PYTHON MODULE INDEX

i