Clique

Release 1.3.1

April 30, 2016

Contents

Manage collections with common numerical component.

CHAPTER 1

Guide

Overview and examples of using the system in practice.

1.1 Introduction

Clique is a library for managing collections that have a common numerical component.

A numerical component is any series of numbers in an item. The item *sc010_020_v001.0005.dpx* has four possible numerical components (**bolded**):

- sc**010**_020_v001.0005.dpx
- sc010_**020**_v001.0005.dpx
- sc010_020_v**001**.0005.dpx
- sc010_020_v001.**0005**.dpx

A common use would be to determine sequences of files on disk. For example, given the following input:

- file.0001.dpx
- file.0002.dpx
- file.0001.jpg
- file.0002.jpg

Clique can automatically assemble two collections:

- file.[index].dpx
- file.[index].jpg

where *[index]* is the commonly changing numerical component.

Read the *Tutorial* to find out more.

1.1.1 Copyright & License

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

1.2 Installation

Installing Clique is simple with pip:

```
$ pip install clique
```

If the Cheeseshop (a.k.a. PyPI) is down, you can also install Clique from one of the mirrors:

```
$ pip install --use-mirrors clique
```

Alternatively, you may wish to download manually from Github where Clique is actively developed.

You can clone the public repository:

```
$ git clone git://github.com/4degrees/clique.git
```

Or download an appropriate tarball or zipball

Once you have a copy of the source, you can embed it in your Python package, or install it into your site-packages:

```
$ python setup.py install
```

1.2.1 Dependencies

- Python >= 2.6, < 3

For testing:

- Pytest >= 2.3.5

1.3 Tutorial

This tutorial gives a good introduction to using Clique.

First make sure that you have Clique *installed*.

Clique revolves around creating collections of items that all have a commonly changing numerical component. Clique itself does not care what the numerical component represents. It could be a frame index for a sequence of files or a version number in a list of versioned files.

The easiest way to create these collections is to `assemble()` them from arbitrary items.

First, import clique:

```
>>> import clique
```

Then define the items to assemble (could be the result of `os.listdir()` for example):

```
>>> items = ['file.0001.jpg', '_cache.txt', 'file.0002.jpg',
...          'foo.1.txt', 'file.0002.dpx', 'file.0001.dpx',
...          'file.0010.dpx', 'scene_v1.ma', 'scene_v2.ma']
```

Finally, assemble them into collections:

```
>>> collections, remainder = clique.assemble(items)
>>> for collection in collections:
...     print repr(collection)
<Collection "scene_v%d.ma [1-2]">
<Collection "file.%04d.dpx [1-2, 10]">
<Collection "file.%04d.jpg [1-2]">
```

Notice how the items `_cache.txt` and `foo.1.txt` didn't form any collections (and were added to `remainder`). This is because `_cache.txt` has no numerical component and was ignored, whilst `foo.1.txt` resulted in a collection with only one item and was filtered out of the result.

The minimum items filter can be altered at assembly time:

```
>>> collections, remainder = clique.assemble(items, minimum_items=1)
>>> for collection in collections:
...     print repr(collection)
<Collection "scene_v%d.ma [1-2]">
<Collection "foo.%d.txt [1]">
<Collection "file.%04d.dpx [1-2, 10]">
<Collection "file.%04d.jpg [1-2]">
```

See also:

There is a full guide to *Assembly* available.

Each collection holds various properties to describe the items it contains:

```
>>> collection = collections[0]
>>> print collection.head
scene_v
>>> print collection.tail
.ma
>>> print collection.padding
0
>>> print collection.indexes
[1, 2]
```

See also:

There is a full guide to *Collections* available.

It is also possible to parse a string (such as that returned from *`Collection.format`*) to create a collection. To do this, use the *`parse()`* function:

```
>>> collection = clique.parse('/path/to/file.%04d.ext [1, 2, 5-10]')
>>> print repr(collection)
<Collection "/path/to/file.%04d.ext [1-2, 5-10]">
```

It is also possible to pass in a different pattern to the default one:

```
>>> collection = clique.parse(
...     '/path/to/file.%04d.ext [1-10] (2, 8)'
...     '{head}{padding}{tail} [{range}] ({holes})'
... )
>>> print repr(collection)
<Collection "/path/to/file.%04d.ext [1, 3-7, 9-10]">
```

1.4 Assembly

As seen in the *Tutorial*, Clique provides the high-level `assemble()` function to support automatically assembling items into relevant *collections* based on a common changing numerical component:

```
>>> import clique
>>> collections, remainder = clique.assemble([
...     'file.0001.jpg', 'file.0002.jpg', 'file.0003.jpg',
...     'file.0001.dpx', 'file.0002.dpx', 'file.0003.dpx'
... ])
>>> print collections
[<Collection "file.%04d.dpx [1-3]">, <Collection "file.%04d.jpg [1-3]">]
```

Note: Any items that are not members of a returned collection can be found in the *remainder* list.

However, as mentioned in the *Introduction*, Clique has no understanding of what a numerical component represents. Therefore, it takes a conservative approach and considers **all** collections with a common changing numerical component as valid. This can lead to surprising results at first:

```
>>> collections, remainder = clique.assemble([
...     'file_v1.0001.jpg', 'file_v1.0002.jpg', 'file_v1.0003.jpg',
...     'file_v2.0001.jpg', 'file_v2.0002.jpg', 'file_v2.0003.jpg'
... ])
>>> print collections
[<Collection "file_v1.%04d.jpg [1-3]">,
 <Collection "file_v2.%04d.jpg [1-3]">,
 <Collection "file_v%d.0001.jpg [1-2]">,
 <Collection "file_v%d.0002.jpg [1-2]">,
 <Collection "file_v%d.0003.jpg [1-2]">]
```

Here, Clique returned more collections that might have been expected, but, as can be seen, they are all valid collections. This is an important feature of Clique - it doesn't attempt to guess. Instead, it is designed to be wrapped easily with domain specific logic to get the results desired.

There are a couple of ways to influence the returned result from the `assemble()` function:

- Pass a *minimum_items* argument.
- Pass custom *patterns*.

1.4.1 Minimum Items

By default, Clique will filter out any collection from the returned result of `assemble()` that has less than two items. This value can be customised per `assemble()` call by passing *minimum_items* as a keyword:

```
>>> print clique.assemble(['file.0001.jpg'])[0]
[]
>>> print clique.assemble(['file.0001.jpg'], minimum_items=1)[0]
[<Collection "file.%04d.jpg [1]">]
```

1.4.2 Patterns

By default, Clique finds all groups of numbers in each item and creates collections that have common *head*, *tail* and *padding* values.

Custom patterns can be used to tailor the process. Pass them as a list of regular expressions (either strings or `re.RegexObject` instances):

```
>>> items = [
...     'file.0001.jpg', 'file.0002.jpg', 'file.0003.jpg',
...     'file.0001.dpx', 'file.0002.dpx', 'file.0003.dpx'
... ])
>>> print clique.assemble(items, patterns=[
...     '\.(?P<index>(?P<padding>0*)\d+)\.\D+\d?$'
... ])[0]
[<Collection "file_v1.%04d.jpg [1-3]">,
 <Collection "file_v2.%04d.jpg [1-3]">]
```

Note: Each custom expression **must** contain the expression from *DIGITS_PATTERN* exactly once. An easy way to do this is using Python's string formatting.

So, instead of:

```
'\.(?P<index>(?P<padding>0*)\d+)\.\D+\d?$'
```

use:

```
'\.{0}\.\D+\d?$'.format(clique.DIGITS_PATTERN)
```

Some common expressions are predefined in the *PATTERNS* dictionary (contributions welcome!):

```
>>> print clique.assemble(items, patterns=[clique.PATTERNS['frames']])[0]
[<Collection "file_v1.%04d.jpg [1-3]">,
 <Collection "file_v2.%04d.jpg [1-3]">]
```

1.4.3 Case Sensitivity

When assembling collections, it is sometimes useful to be able to specify whether the case of the items should be important or not. For example, "file.0001.jpg" and "FILE.0002.jpg" could be identified as part of the same collection or not.

By default the assembly is case sensitive, but this can be controlled by setting the *case_sensitive* argument:

```
>>> items = ['file.0001.jpg', 'FILE.0002.jpg', 'file.0003.jpg']
>>> print clique.assemble(items, case_sensitive=False)
[<Collection "file.%04d.jpg [1-3]">], []
>>> print clique.assemble(items, case_sensitive=True)
[<Collection "file.%04d.jpg [1, 3]">], ['FILE.0002.jpg']
```

A common use case might be to ignore case sensitivity when on a Windows or Mac machine:

```
>>> import sys
>>> clique.assemble(
...     items, case_sensitive=sys.platform not in ('win32', 'darwin')
... )
```

1.5 Collections

A collection holds items that all have a single common numerical component, whose value differs between each item.

Each collection comprises three main attributes:

- *head* - The common leading part of each item.
- *tail* - The common trailing part of each item.
- *padding* - The width of the index (to be padded to with zeros).

Given items such as:

- file.0001.jpg
- file.0002.jpg

The *head* would be `file.`, the *tail* `.jpg` and the *padding* `4`.

Note: If the numerical component is unpadded then the *padding* would be 0 and a variable index width supported.

A collection can be manually created using the *Collection* class:

```
>>> import clique
>>> collection = clique.Collection(head='file.', tail='.jpg', padding=4)
```

1.5.1 Adding & Removing Items

Items can then be *added* to the collection:

```
>>> collection.add('file.0001.jpg')
```

If an item does not match the collection's expression a *CollectionError* is raised:

```
>>> collection.add('file.0001.dpx')
CollectionError: Item does not match collection expression.
```

Whether an item matches the collection expression can be tested ahead of time if desired using *match()*:

```
>>> print collection.match('file.0002.jpg')
<_sre.SRE_Match object at 0x0000000003710D78>
>>> print collection.match('file.0002.dpx')
None
```

To remove an item:

```
>>> collection.remove('file.0001.jpg')
```

If the item is not present, a *CollectionError* is raised:

```
>>> collection.remove('file.0001.jpg')
CollectionError: Item not present in collection.
```

1.5.2 Accessing Items

To access items in the collection, iterate over it:

```
>>> collection.add('file.0001.jpg')
>>> collection.add('file.0002.jpg')
>>> for item in collection:
...     print item
```

```
file.0001.jpg
file.0002.jpg
```

Note: A collection may be sparse and so is not directly indexable. If you need to access an item by index, convert the collection to a list:

```
>>> print list(collection)[-1]
file.0002.jpg
```

1.5.3 Manipulating Indexes

Internally, Clique does not store the items directly, but rather just the properties to recreate the items (*head*, *tail*, *padding*). In addition it holds a sorted set of indexes present in the collection.

This set of indexes can be manipulated directly to perform the equivalent of adding and removing items (perhaps in bulk).

```
>>> print collection.indexes
[1, 2]
>>> collection.indexes.update([2, 3, 4])
>>> for item in collection:
...     print item
file.0001.jpg
file.0002.jpg
file.0003.jpg
file.0004.jpg
```

Note: It is not possible to assign a new index set directly:

```
>>> collection.indexes = set([1, 2, 3])
AttributeError: Cannot set attribute defined as unsettable.
```

Instead, first clear and update the set as required:

```
>>> collection.indexes.clear()
>>> collection.indexes.update(set([1, 2, 3])
```

1.5.4 Formatting

It is useful to express a collection as a string that represents the collection expression and ranges in a standard way. Clique supports basic formatting of a collection through its *format()* method:

```
>>> collection = clique.Collection('file.', '.jpg', 4, indexes=set([1, 2]))
>>> print collection.format()
file.%04d.jpg [1-2]
```

The *format()* method can be passed an alternative pattern if required:

```
>>> print collection.format('{head}[index]{tail}')
file.[index].jpg
```

The passed pattern should match the formatting rules of Python's standard string formatter and will have the following keyword variables available to it:

- *:term:'head'* - Common leading part of the collection.
- *:term:'tail'* - Common trailing part of the collection.
- *:term:'padding'* - Padding value in `%0d` format.
- *range* - Total range in the form `start-end`
- *ranges* - Comma separated ranges of indexes.
- *holes* - Comma separated ranges of missing indexes.

1.5.5 Structure

Clique makes it easy to get further information about the structure of a collection and act on that structure.

To check if a collection contains items that make up a *contiguous* sequence use *is_contiguous()*:

```
>>> collection = clique.Collection('file.', '.jpg', 4)
>>> collection.indexes.update([1, 2, 3, 4, 5])
>>> print collection
file.%04d.jpg [1-5]
>>> print collection.is_contiguous()
True
>>> collection.indexes.discard(3)
>>> print collection
file.%04d.jpg [1-2, 4-5]
>>> print collection.is_contiguous()
False
```

To access the missing indexes in a non-*contiguous* collection use the *holes()* method (which returns a new *Collection*):

```
>>> missing = collection.holes()
>>> print missing.indexes
[3]
```

To separate a non-*contiguous* collection into a number of *contiguous* collections use the *separate()* method:

```
>>> subcollections = collection.separate()
>>> for subcollection in subcollections:
...     print subcollection
file.%04d.jpg [1-2]
file.%04d.jpg [4-5]
```

And to merge compatible collections into another use the *merge()* method:

```
>>> collection_a = clique.Collection('file.', '.jpg', 4, set([1, 2]))
>>> collection_b = clique.Collection('file.', '.jpg', 4, set([4, 5]))
>>> print collection_a.indexes
[1, 2]
>>> collection_a.merge(collection_b)
>>> print collection_a.indexes
[1, 2, 4, 5]
```

Note: The collection being merged into is modified in-place, whilst the collection being merged is left unaltered.

A collection can be tested for compatibility using the *is_compatible()* method:

```
>>> collection_a = clique.Collection('file.', '.jpg', 4, set([1, 2]))
>>> collection_b = clique.Collection('file.', '.jpg', 4, set([4, 5]))
>>> collection_c = clique.Collection('file.', '.dpx', 4, set([4, 5]))

>>> print collection_a.is_compatible(collection_b)
True
>>> print collection_a.is_compatible(collection_c)
False
```

CHAPTER 2

Reference

API reference providing details on the actual code.

2.1 clique

clique.**DIGITS_PATTERN** = '(?P<index>(?P<padding>0*)\\d+)'
Pattern for matching an index with optional padding.

clique.**PATTERNS** = {'frames': '\\.(?P<index>(?P<padding>0*)\\d+)\\.\\D+\\d?$', 'versions': 'v(?P<index>(?P<padding>0*)\\
Common patterns that can be passed to *assemble()*.

clique.**assemble**(*iterable*, *patterns=None*, *minimum_items=2*, *case_sensitive=True*)
Assemble items in *iterable* into discreet collections.

patterns may be specified as a list of regular expressions to limit the returned collection possibilities. Use this when interested in collections that only match specific patterns. Each pattern must contain the expression from *DIGITS_PATTERN* exactly once.

A selection of common expressions are available in *PATTERNS*.

Note: If a pattern is supplied as a string it will be automatically compiled to a `re.RegexObject` instance for convenience.

When *patterns* is not specified, collections are formed by examining all possible groupings of the items in *iterable* based around common numerical components.

minimum_items dictates the minimum number of items a collection must have in order to be included in the result. The default is 2, filtering out single item collections.

If *case_sensitive* is False, then items will be treated as part of the same collection when they only differ in casing. To avoid ambiguity, the resulting collection will always be lowercase. For example, "item.0001.dpx" and "Item.0002.dpx" would be part of the same collection, "item.%04d.dpx".

Note: Any compiled *patterns* will also respect the set case sensitivity.

Return tuple of two lists (collections, remainder) where 'collections' is a list of assembled *Collection* instances and 'remainder' is a list of items that did not belong to any collection.

clique.**parse**(*value*, *pattern='{head}{padding}{tail} [{ranges}]'*)
Parse *value* into a *Collection*.

Use *pattern* to extract information from *value*. It may make use of the following keys:

- *head* - Common leading part of the collection.
- *tail* - Common trailing part of the collection.
- *padding* - Padding value in `%0d` format.
- *range* - Total range in the form `start-end`.
- *ranges* - Comma separated ranges of indexes.
- *holes* - Comma separated ranges of missing indexes.

Note: *holes* only makes sense if *range* or *ranges* is also present.

2.1.1 collection

class clique.collection.**Collection**(*head*, *tail*, *padding*, *indexes=None*)
Bases: object

Represent group of items that differ only by numerical component.

__init__(*head*, *tail*, *padding*, *indexes=None*)
Initialise collection.

head is the leading common part whilst *tail* is the trailing common part.

padding specifies the "width" of the numerical component. An index will be padded with zeros to fill this width. A *padding* of zero implies no padding and width may be any size so long as no leading zeros are present.

indexes can specify a set of numerical indexes to initially populate the collection with.

Note: After instantiation, the `indexes` attribute cannot be set to a new value using assignment:

```
>>> collection.indexes = [1, 2, 3]
AttributeError: Cannot set attribute defined as unsettable.
```

Instead, manipulate it directly:

```
>>> collection.indexes.clear()
>>> collection.indexes.update([1, 2, 3])
```

---

head
Return common leading part.

tail
Return common trailing part.

match(*item*)
Return whether *item* matches this collection expression.

If a match is successful return data about the match otherwise return None.

add(*item*)
Add *item* to collection.

raise *CollectionError* if *item* cannot be added to the collection.

remove(*item*)
Remove *item* from collection.

raise *CollectionError* if *item* cannot be removed from the collection.

format(*pattern='{head}{padding}{tail} [{ranges}]'*)
Return string representation as specified by *pattern*.

Pattern can be any format accepted by Python's standard format function and will receive the following keyword arguments as context:

- *head* - Common leading part of the collection.
- *tail* - Common trailing part of the collection.
- *padding* - Padding value in `%0d` format.
- *range* - Total range in the form `start-end`
- *ranges* - Comma separated ranges of indexes.
- *holes* - Comma separated ranges of missing indexes.

is_contiguous()
Return whether entire collection is contiguous.

holes()
Return holes in collection.

Return *Collection* of missing indexes.

is_compatible(*collection*)
Return whether *collection* is compatible with this collection.

To be compatible *collection* must have the same head, tail and padding properties as this collection.

merge(*collection*)
Merge *collection* into this collection.

If the *collection* is compatible with this collection then update indexes with all indexes in *collection*.

raise *CollectionError* if *collection* is not compatible with this collection.

separate()
Return contiguous parts of collection as separate collections.

Return as list of *Collection* instances.

2.1.2 error

Custom error classes.

exception clique.error.**CollectionError**
Bases: exceptions.Exception

Raise when a collection error occurs.

2.1.3 sorted_set

class clique.sorted_set.**SortedSet**(*iterable=None*)
Bases: _abcoll.MutableSet

Maintain sorted collection of unique items.

__init__(*iterable=None*)
Initialise with items from *iterable*.

add(*item*)
Add *item*.

discard(*item*)
Remove *item*.

update(*iterable*)
Update items with those from *iterable*.

2.1.4 descriptor

class clique.descriptor.**Unsettable**(*label*)
Bases: object

Prevent standard setting of property.

Example:

```
>>> class Foo(object):
...
...     x = Unsettable('x')
...
...     def __init__(self):
...         self.__dict__['x'] = True
...
>>> foo = Foo()
>>> print foo.x
True
>>> foo.x = False
AttributeError: Cannot set attribute defined as unsettable.
```

__init__(*label*)
Initialise descriptor with property *label*.

label should match the name of the property being described:

```
        x = Unsettable('x')
```

CHAPTER 3

Glossary

contiguous When all items in a collection are sequential with no missing indexes. For example, *1, 2, 3* is contiguous whilst *1, 3* is not.

head The common leading part of items in a collection. For example, the items *file.0001.jpg*, *file.0002.jpg*, *file.0003.jpg* have a head value of *file.*

padding The width of the numerical index in a collection. Each item's index will be padded with zeroes to match this width. A padding of 4 would result in *1* becoming *0001*. A padding of 0 means no width is defined and an index can be any width so long as it has no preceding zeroes.

tail The common trailing part of items in a collection. For example, the items *file.0001.jpg*, *file.0002.jpg*, *file.0003.jpg* have a tail value of *.jpg*

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

c

d

e

s

Symbols

A

C

D

F

H

I

M

P

R

S

T

U