CherryPicker Documentation

Release 0.1.0

big-o@users.noreply.github.com

Jul 16, 2019

Contents

Flatten complex data.

`cherrypicker` aims to make common ETL tasks (filtering data and restructuring it into flat tables) easier, by taking inspiration from jQuery and applying it in a Pythonic way to generic data objects.

```
pip install cherrypicker
```

`cherrypicker` provides a chainable filter and extraction interface to allow you to easily pick out objects from complex structures and place them in a flat table. It fills a similar role to jQuery in JavaScript, enabling you to navigate complex structures without the need for lots of complex nested for loops or list comprehensions.

Behold. . .

```
>>> from cherrypicker import CherryPicker
>>> import json
>>> data = json.load(open('climate.json'))
>>> picker = CherryPicker(data)
>>> picker['id', 'city'].get()
[[1, 'Amsterdam'], [2, 'Athens'], [3, 'Atlanta GA'], ...]
```

This example is equivalent to the list comprehension `[[item['id'], item['city']] for item in data]`. `cherrypicker` really starts to become useful when you combine it with filtering:

```
>>> picker(city='B*')['id', 'city'].get()
[[6, 'Bangkok'], [7, 'Barcelona'], [8, 'Beijing'], ...]
```

The equivalent list comprehension would be: `[[item['id'], item['city']] for item in data if item['city'].startswith('B')]`. As you can see, `CherryPicker` does filtering and extraction with chained operations rather than list comprehensions. *Filtering* is done with parentheses `()` and *extraction* is done with square brackets `[]`. Chaining can make it easier to build complex operations:

```
>>> picker(city='B*')['info'](
...     population=lambda n: n > 2000000, area=lambda a: a < 2000, how='all'
... )['area', 'population'].get()
[[1568, 8300000], [891, 3700000], [203, 2800000]]
```

This is getting too unwieldy for list comprehensions already; to achieve the example above in another way we may wish to use a for loop:

```
table = []
for item in data:
    if item['city'].startswith('B'):
        info = item['info']
        if info['population'] > 2000000 and info['area'] < 2000:
            table.append(info['area'], info['population'])
```

Without `cherrypicker`, the amount of code we need to write increases pretty rapidly! There are many different types of filtering predicate you can use with `cherrypicker`, including exact matches, wildcards, regex and custom functions. Read all about them in the *Filtering* documentation.

Of course, it would be nice if we could extract data in the example above from both the base level and the `info` sub-level of each item and put them into a flat table, ready to load into your favourite data analysis package. We can do this in `cherrypicker` with *`cherrypicker.CherryPickerMapping.flatten()`*. Let's say that each item in our data list has a city name and a list of average low/high temperatures for each month of the year:

```
[
    {
        "id": 1,
```

(continues on next page)

(continued from previous page)

```
        "city": "Amsterdam",
        "country": "Netherlands",
        "monthlyAvg": [
            {
                "high": 7,
                "low": 3,
                "dryDays": 19,
                "snowDays": 4,
                "rainfall": 68
            },
            {
                "high": 6,
                "low": 3,
                "dryDays": 13,
                "snowDays": 2,
                "rainfall": 47
            },
            ...
        ]
    }
]
```

By flattening the data before filtering/extracting, we can get the name and monthly temperatures alongside each other:

```
>>> picker.flatten(monthlyAvg_0_high=lambda tmp: tmp > 30)['city', 'monthlyAvg_0_high
↪'].get()
[['Bangkok', 33], ['Brasilia', 31], ['Ho Chi Minh City', 33], ...]
```

```
>>> picker.flatten(monthlyAvg_0_high=lambda tmp: tmp < 0)['city', 'monthlyAvg_0_high
↪'].get()
[['Calgary', -1], ['Montreal', -4], ['Moscow', -4], ...]
```

One final point to note is that `cherrypicker` understands data by looking at its *interfaces* rather than its *types*. This means that it isn't just limited to JSON data: as long as it can act like a dict or list, you can start cherrypicking from it!

CHAPTER 1

Parallel Processing

As well as making complex queries easier, `CherryPicker` also allows you to easily use parallel processing to crunch through large datasets quickly:

```
>>> picker = CherryPicker(data, n_jobs=4)
>>> picker(city='B*')['id', 'city'].get()
```

Everything is the same as before, except you supply an *n_jobs* parameter to specify the number of CPUs you wish to use (a value of *-1* will mean all CPUs are used).

Note that for small datasets, you will probably get better performance without parallel processing, as the benefits of using multiple CPUs will be outweighed by the overhead of setting up multiple processes. For large datasets with long lists though, parallel processing can significantly speed up your operations.

1.1 Filtering

`CherryPicker` objects navigate your data by the following rules:

- If it implements the `collections.abc.Mapping` interface, treat it like a `dict`;
- Otherwise if it implements the `collections.abc.Iterable` interface, treat it like a `list`;
- Otherwise it is treated as a leaf node (*i.e.* an end point).

You apply filters to your data by providing *predicates* in parentheses after you have navigated to the data you want, for example:

```
>>> picker = CherryPicker(data)
>>> picker(name='Alice')
<CherryPickerIterable(list, len=1)>
```

. . . applied the predicate `name='Alice'` to the root data node. There were twelve items in the data that matched this filter. To see the actual data that was extracted, use the `cherrypicker.CherryPicker.get()` method:

```
>>> picker(name='Alice').get()
[{'name': 'Alice', 'age': 20}]
```

Filters behave slightly differently depending on what type of data you have at the point you apply them:

- If the data is `dict`-like, each predicate will be applied to the value obtained by the key matching the predicate parameter. In the example above, the value for the key `name` will be checked, and if it is `'Alice'`, the filter has passed and the object will be returned. If the filter fails, the default item (which defaults to a leaf containing `None`) is returned instead.
- If the data is `list`-like, the filter will be applied to each child. A new `list`-like node will be returned containing only the matching items.

1.1.1 Combining predicates

Multiple predicates can be applied in a single filter. The *how* parameter determines the logic used to combine them. If *how* is `'all'` (which is the default), all predicates must match. If *how* is `'any'`, only one predicate needs to match for the filter to pass.

1.1.2 Types of predicate

The value supplied for each predicate term determines the kind of test that is performed:

- If the predicate is a string, one of the following checks will be done:
  - If *allow_wildcards=True* and the string contains a wildcard character as defined by `fnmatch.fnmatchcase()`, then a wildcard match is performed.
  - If *case_sensitive=False*, a case-insensitive string comparison will be made.
  - If *regex=True* then the string will be compiled into a regular expression. A `re.fullmatch()` test will be performed. If *case_sensitive* is also *False*, the regex test will be case-insensitive.
  - Otherwise, only an exact match is accepted.
- If the predicate is a compiled regular expression pattern, a `re.fullmatch()` test will be performed.
- If the predicate is a callable function or lambda, the function will be applied to the value being tested. This function should take in a single parameter (the value) and return something that evaluates to `True` or `False`.

1.1.3 API

`CherryPickerTraversable.`**`filter`**(*how='all', allow_wildcards=True, case_sensitive=True, regex=False, opts=None, **predicates*)

Return a filtered view of the child nodes. This method is usually accessed via `CherryPicker.__call__()`

For an object with a mappable interface, this will return the object itself if it matches the predicates according to the rules specified.

For an object with an iterable but not a mappable interface, a collection of child objects matching the predicates according to the rules specified will be returned.

This method is not implemented for leaf nodes and will cause an error to be raised.

Example

Find any items with a name of `Alice`:

```
>>> picker(name='Alice')
```

Find any items with a name of `Alice` and an age of 20:

```
>>> picker(name='Alice', age=20)
```

Find any items with a name of `Alice` *or* an age of 20:

```
>>> picker(name='Alice', age=20, how='any')
```

Find any items with a name of `Alice` and an age of 20 or more:

```
>>> picker(name='Alice', age=lambda a: a >= 20)
```

Find any items with a name beginning with `Al`:

```
>>> picker(name='Al*')
```

Find any items with a name beginning with `Al` or `al`:

```
>>> picker(name='Al*', case_sensitive=False)
```

Find any items with a name of `Al*`:

```
>>> picker(name='Al*', allow_wildcards=False)
```

Find any items with a name matching a particular pattern (these two lines are equivalent):

```
>>> picker(name=r'^(?:Alice|Bob)$', regex=True, case_sensitive=False)
>>> picker(name=re.compile(r'^(?:Alice|Bob)$', re.I))
```

Parameters

- **how** (*str.*) – The rule to be applied to predicate matching. May be one of ('all', 'any').
- **allow_wildcards** (*bool, default = True.*) – If True, special characters (`*`, `?`, `[]`) in any string predicate values will be treated as wildcards according to `fnmatch.fnmatchcase()`.
- **case_sensitive** (*bool, default = True.*) – If True, any comparisons to strings or uncompiled regular expressions will be case sensitive.
- **regex** (*bool, default = False.*) – If True, any string comparisons will be reinterpreted as regular expressions. If `case_sensitive` is False, they will be case-insensitive patterns. For more complex regex options, omit this parameter and provide pre-compiled regular expression patterns in your predicates instead. All regular expressions will be compared to string values using a full match.
- **predicates** (*str, regular expression or Callable.*) – Keyword arguments where the keys are the object keys used to get the comparison value, and the values are either a value to compare, a regular expression to perform a full match against, or a callable function that takes a single value as input and returns something that evaluates to True if the value passes the predicate, or False if it does not.

Returns If this is a mappable object, the object itself if it passes the predicates. If not and this is an iterable object, a collection of children that pass the predicates.

Return type *CherryPicker*.

1.2 Extraction

When you have navigated to the data node(s) you care about, you can extract data from them into flat tables. These tables will always contain the values you have requested in a `list` or nested lists. Lists are used to improve compatability with `pandas` and `numpy`.

Much like *filtering*, extraction differs depending on the type of data node you are operating on:

- If the data is `dict`-like, values will be extracted from all the keys provided into a flat list.
- If the data is `list`-like, data extraction will be delegated to each item in the collection, and the results returned in another list.

Data is extracted with the square brackets (`[]`) operator. When you extract data, the results are wrapped up in another `CherryPicker` object (this is to enable the chaining of operations). At any stage in your cherry picking, you can get down to the raw data with the `CherryPicker.get()` operator:

```
>>> picker = CherryPicker(data)
>>> picker[0]['id', 'city']
<CherryPickerIterable(list, len=2)>
>>> picker[0]['id', 'city'].get()
[1, 'Amsterdam']
```

1.2.1 Navigating lists

Lists (or any `list`-like object) is a little more complicated than it first seems. What if you want to get the first item of a list? Well that's easy:

```
>>> picker = CherryPicker(mylist)
>>> picker
<CherryPickerIterable(list, len=105)>
>>> picker[0]
<CherryPickerMapping(dict)>
```

It's just like working with a normal list. Just provide the index of the item you want and your cherry picker will get it for you. Slices are also accepted if you want multiple items. If you want to drill down and extract items from each item in the list, that's also easy enough:

```
>>> picker['city']
<CherryPickerIterable(list, len=105)>
```

You can see here that this time the cherry picker has given you a list of results. That's because it's extracted the *city* from each item in the list. Cherry pickers are usually smart enough to know when you want to grab something from the list itself vs. grab something from the items in the list.

But what if `mylist` was actually a list of lists, and you actually wanted to get the first item of each list? Things aren't as clear now, because your picker will assume that `picker[0]` means grab the entire first item only. In this case, you must give your picker an extra hint. For `list`-like objects, if you provide an `int` or `slice`-like parameter, you can also provide an optional second boolean parameter known as the *propagate* flag. If this flag is set to True, the picker will apply the index to each child node, regardless of what type it is:

```
>>> mynestedlist = [['Alice', 20], ['Bob', 34], ...]
>>> picker = CherryPicker(mynestedlist)
>>> picker[0].get()
['Alice', 20]
```

(continues on next page)

(continued from previous page)

```
>>> picker[0, True].get()
['Alice', 'Bob', ...]
```

In the first command, the first item in the list (another list of length 2) is obtained. In the second command, the *propagate* flag is set, so we instead grab the first item of each child instead.

1.3 API Reference

class cherrypicker.**CherryPicker**(*obj*, *on_missing='ignore'*, *on_error='ignore'*, *on_leaf='raise'*, *leaf_types=(<class 'str'>, <class 'bytes'>)*, *default=None*, *n_jobs=None*)

Reduces nestings of iterable and mappable objects into flat tables.

The CherryPicker class allows you to apply chained filter and extract operations to an object with complex structure. All the cherry picker uses to navigate your object is iterable and mapping interfaces. Anything without either of those interfaces (or a string) is treated as a leaf node.

Each chained operation will return a new `CherryPicker` which wraps the resulting data from that operation. To get the wrapped data back, use the `CherryPicker.get()` method.

Parameters

- **obj** (`object.`) – The data to operate on.
- **on_missing** (str, default = `ignore.`) – Action to perform when trying to get an attribute that doesn't exist from an object with a Mapping interface. `ignore` will do nothing, `raise` will raise an `AttributeError`.
- **on_error** (str, default = `ignore`) – Action to perform if an error occurs during filtering. `ignore` will just mean the filter operation returns False, and `raise` will mean the error is raised.
- **on_leaf** (str, default = `raise.`) – Action to perform when calling `__getitem__()` on a leaf node. `raise` will cause a `cherrypicker.exceptions.LeafError`` to be raised. `get` will return the result of `__getitem__()` on the wrapped item.
- **leaf_types** – By default, anything doesn't have an Iterable or Mapping interface will be treated as a leaf. Any classes specifed in this parameter will also be treated as leaves regardless of any interfaces they conform to. `leaf_types` may be a class, a method that resolves to True if an object passed to it should be treated as a leaf, or a tuple of classes/methods.
- **default** (`object, default = None`) – The item to return when extracting an attribute that does not exist from an object.
- **n_jobs** (`int, default = None`) – The maximum number of parallel processes to run when performing operations on iterable objects. If n_jobs > 1 then the iterable will be processed in parallel batches. If n_jobs = -1, all the CPUs are used. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used. See `joblib.Parallel` for more details on this parameter.

Examples

Data extraction may be done with the getitem interface. Let's say we have a list of objects and we want to get a flat list of the `name` attributes for each item in the list:

```
>>> data = [ { 'name': 'Alice', 'age': 20}, { 'name': 'Bob', 'age': 30 } ]
>>> picker = CherryPicker(data)
>>> picker['name'].get()
['Alice', 'Bob']
```

We can also request multiple attributes for each item to produce a flat table:

```
>>> data = [ { 'name': 'Alice', 'age': 20}, { 'name': 'Bob', 'age': 30 } ]
>>> picker = CherryPicker(data)
>>> picker['name', 'age'].get()
[['Alice', 20], ['Bob', 30]]
```

Filter operations are applied with parentheses. For example, to get every `name` attribute from each item in a list called `data`:

```
>>> data = [ { 'name': 'Alice', 'age': 20}, { 'name': 'Bob', 'age': 30 } ]
>>> picker = CherryPicker(data)
>>> picker(name='Alice')['age'].get()
[30]
```

Multiple filters may be provided:

```
>>> data = [ { 'name': 'Alice', 'age': 20}, { 'name': 'Bob', 'age': 30 } ]
>>> picker = CherryPicker(data)
>>> picker(name='Alice' age=lambda x: x>10, how='any').get()
[{'name': 'Alice', 'age': 20}, {'name': 'Bob', 'age': 30}]
```

Filters can also be chained:

```
>>> data = [ { 'name': 'Alice', 'age': 20}, { 'name': 'Bob', 'age': 30 } ]
>>> picker = CherryPicker(data)
>>> picker(age=lambda x: x>10)(name='B*')['name'].get()
['Bob']
```

See `CherryPicker.filter()` for more filtering options.

get()
  Obtain the original data that this object wraps.

parent
  Get the parent or iterable of parents.

parents
  Alias for `parent()`.

class cherrypicker.**CherryPickerTraversable**(*obj*, *on_missing='ignore'*, *on_error='ignore'*, *on_leaf='raise'*, *leaf_types=(<class 'str'>, <class 'bytes'>)*, *default=None*, *n_jobs=None*)
  Abstract class for traversable (mappable and/or iterable) nodes.

  filter(*how='all'*, *allow_wildcards=True*, *case_sensitive=True*, *regex=False*, *opts=None*, ***predicates*)
    Return a filtered view of the child nodes. This method is usually accessed via `CherryPicker.__call__()`

    For an object with a mappable interface, this will return the object itself if it matches the predicates according to the rules specified.

For an object with an iterable but not a mappable interface, a collection of child objects matching the predicates according to the rules specified will be returned.

This method is not implemented for leaf nodes and will cause an error to be raised.

Example

Find any items with a name of `Alice`:

```
>>> picker(name='Alice')
```

Find any items with a name of `Alice` and an age of 20:

```
>>> picker(name='Alice', age=20)
```

Find any items with a name of `Alice` *or* an age of 20:

```
>>> picker(name='Alice', age=20, how='any')
```

Find any items with a name of `Alice` and an age of 20 or more:

```
>>> picker(name='Alice', age=lambda a: a >= 20)
```

Find any items with a name beginning with `Al`:

```
>>> picker(name='Al*')
```

Find any items with a name beginning with `Al` or `al`:

```
>>> picker(name='Al*', case_sensitive=False)
```

Find any items with a name of `Al*`:

```
>>> picker(name='Al*', allow_wildcards=False)
```

Find any items with a name matching a particular pattern (these two lines are equivalent):

```
>>> picker(name=r'^(?:Alice|Bob)$', regex=True, case_sensitive=False)
>>> picker(name=re.compile(r'^(?:Alice|Bob)$', re.I))
```

Parameters

- **how** (*str.*) – The rule to be applied to predicate matching. May be one of ('all', 'any').
- **allow_wildcards** (*bool, default = True.*) – If True, special characters (`*`, `?`, `[]`) in any string predicate values will be treated as wildcards according to `fnmatch.fnmatchcase()`.
- **case_sensitive** (*bool, default = True.*) – If True, any comparisons to strings or uncompiled regular expressions will be case sensitive.
- **regex** (*bool, default = False.*) – If True, any string comparisons will be reinterpreted as regular expressions. If `case_sensitive` is False, they will be case-insensitive patterns. For more complex regex options, omit this parameter and provide pre-compiled regular expression patterns in your predicates instead. All regular expressions will be compared to string values using a full match.
- **predicates** (*str, regular expression or Callable.*) – Keyword arguments where the keys are the object keys used to get the comparison value, and the values are either a value to compare, a regular expression to perform a full match against, or a

callable function that takes a single value as input and returns something that evaluates to True if the value passes the predicate, or False if it does not.

Returns If this is a mappable object, the object itself if it passes the predicates. If not and this is an iterable object, a collection of children that pass the predicates.

Return type `CherryPicker`.

class `cherrypicker.`**`CherryPickerIterable`**(*obj*, *on_missing='ignore'*, *on_error='ignore'*, *on_leaf='raise'*, *leaf_types=(<class 'str'>, <class 'bytes'>)*, *default=None*, *n_jobs=None*)

A collection of objects to be cherry picked.

`keys`(*peek=5*)

Parameters **`peek`** (`int`, `optional`) – The maximum number of items in the iterable to inspect in order to ascertain what all possible keys are. If None, all items are inspected.

Returns A view of the keys that exist in *all* items that were previewed. Individual items may have other keys, but they will not be returned unless all the other items inspected also have those keys.

Return type list

class `cherrypicker.`**`CherryPickerMapping`**(*obj*, *on_missing='ignore'*, *on_error='ignore'*, *on_leaf='raise'*, *leaf_types=(<class 'str'>, <class 'bytes'>)*, *default=None*, *n_jobs=None*)

A mappable (key->value pairs) object to be cherry picked from.

`flatten`

Flatten down the object so that all of its values are leaf nodes.

`items`(*peek=None*)

Parameters **`peek`** (`object`, `optional`) – Not used.

Returns A view of the object's items.

Return type list

`keys`(*peek=None*)

Parameters **`peek`** (`object`, `optional`) – Not used.

Returns A view of the object's keys.

Return type list

`values`(*peek=None*)

Parameters **`peek`** (`object`, `optional`) – Not used.

Returns A view of the object's values.

Return type list

class `cherrypicker.`**`CherryPickerLeaf`**(*obj*, *on_missing='ignore'*, *on_error='ignore'*, *on_leaf='raise'*, *leaf_types=(<class 'str'>, <class 'bytes'>)*, *default=None*, *n_jobs=None*)

A non-traversable node (an end-point).

This class cannot perform filter or extract operations; it only exists to return a result (with `get()`).

1.4 Indices and tables

- genindex
- modindex
- search

Index