Cawdrey

Release 0.1.7

Dominic Davis-Foster

Jul 28, 2020

CONTENTS

Several useful custom dictionaries for Python

Docs	
Tests	
PyPI	
Anaconda	
Activity	
Other	

CHAPTER

ONE

CONTENTS

- `frozendict`: An immutable dictionary that cannot be changed after creation.
- `FrozenOrderedDict`: An immutable `OrderedDict` where the order of keys is preserved, but that cannot be changed after creation.
- `AlphaDict`: A `FrozenOrderedDict` where the keys are stored in alphabetical order.
- `bdict`: A dictionary where *key, value* pairs are stored both ways round.

This package also provides two base classes for creating your own custom dictionaries:

- `FrozenBase`: An Abstract Base Class for Frozen dictionaries.
- `MutableBase`: An Abstract Base Class for mutable dictionaries.

CHAPTER

TWO

OTHER DICTIONARY PACKAGES

If you're looking to unflatten a dictionary, such as to go from this:

```
{'foo.bar': 'val'}
```

to this:

```
{'foo': {'bar': 'val'}}
```

check out unflatten, flattery or morph to accomplish that.

indexed provides an OrederedDict where the values can be accessed by their index as well as by their keys.

There's also python-benedict, which provides a custom dictionary with **keylist/keypath** support, **I/O** shortcuts (`Base64`, `CSV`, `JSON`, `TOML`, `XML`, `YAML`, `pickle`, `query-string`) and many **utilities**.

CHAPTER

THREE

INSTALLATION

from PyPI

```
$ python3 -m pip install cawdrey --user
```

from Anaconda

First add the required channels

```
$ conda config --add channels http://conda.anaconda.org/domdfcoding
$ conda config --add channels http://conda.anaconda.org/conda-forge
```

Then install

```
$ conda install cawdrey
```

from GitHub

```
$ python3 -m pip install git+https://github.com/domdfcoding/cawdrey@master --user
```

3.1 AlphaDict

3.1.1 About

3.1.2 Usage

3.1.3 API Reference

class cawdrey.**AlphaDict**(*seq=None*, ***kwargs*)

Initialize an immutable, Alphabetised dictionary. The signature is the same as regular dictionaries.

dict() -> new empty AlphaDict

dict(mapping) -> new AlphaDict initialized from a mapping object's (key, value) pairs

dict(iterable) -> new AlphaDict initialized as if via:

```
d = {}
for k, v in iterable:
    d[k] = v
```

dict(**kwargs) -> new AlphaDict initialized with the name=value pairs in the keyword argument list. For example:

```
dict(one=1, two=2)
```

__abstractmethods__ = frozenset({})

__annotations__ = {'dict_cls': typing.Union[typing.Type, NoneType]}

__args__ = None

__contains__(*key*)

Return type Any

__copy__(**args*, ***kwargs*)

__eq__(*other*)
Return self==value.

__extra__ = None

__getitem__(*key*)

Return type Any

__hash__()
Return hash(self).

Return type int

__init__(*seq=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()

__len__()

Return type int

__module__ = 'cawdrey.alphadict'

static __new__(*cls*, **args*, ***kwds*)
Create and return a new object. See help(type) for accurate signature.

__next_in_mro__
alias of builtins.object

__orig_bases__ = (cawdrey.frozenordereddict.FrozenOrderedDict[~KT, ~VT],)

__origin__ = None

__parameters__ = (~KT, ~VT)

__repr__()
Return repr(self).

Return type str

__reversed__ = None

__slots__ = ()

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__tree_hash__ = -9223366108873949583

__weakref__
list of weak references to the object (if defined)

copy(**args*, ***kwargs*)
Return a copy of the *FrozenOrderedDict*.

Parameters

- **args** –
- **kwargs** –

Returns

Return type

dict_cls
alias of collections.OrderedDict

classmethod fromkeys(**args*, ***kwargs*)
Returns a new dict with keys from iterable and values equal to value.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

values() → an object providing a view on D's values

3.2 bdict

3.2.1 About

3.2.2 Usage

3.2.3 API Reference

class cawdrey.**bdict**(*seq=None*, ***kwargs*)
Returns a new dictionary initialized from an optional positional argument, and a possibly empty set of keyword arguments.

Each key:value pair is entered into the dictionary in both directions, so you can perform lookups with either the key or the value.

If no positional argument is given, an empty dictionary is created.

If a positional argument is given and it is a mapping object, a dictionary is created with the same key-value pairs as the mapping object. Otherwise, the positional argument must be an iterable object. Each item in the iterable must itself be an iterable with exactly two objects. The first object of each item becomes a key in the new dictionary, and the second object the corresponding value.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument.

If an attempt is made to add a key or value that already exists in the dictionary a ValueError will be raised

Keys or values of None, True and False will be stored internally as "_None", "_True" and "_False" respectively

Based on https://stackoverflow.com/a/1063393 by https://stackoverflow.com/users/9493/brian

Improved May 2020 with suggestions from https://treyhunner.com/2019/04/why-you-shouldnt-inherit-from-list-and-dict-in-python/

__abstractmethods__ = frozenset({})

__contains__(*key*)

Return type bool

__delitem__(*key*)

__eq__(*other*)
Return self==value.

__getitem__(*key*)

Return type Any

__hash__ = None

__init__(*seq=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()

__len__()

__module__ = 'cawdrey._bdict'

__repr__()
Return repr(self).

__reversed__ = None

__setitem__(*key*, *val*)

__slots__ = ()

classmethod __subclasshook__(*C*)
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

clear() → None. Remove all items from D.

copy()

classmethod fromkeys(*iterable*, *value=None*)

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update(*[E]*, ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

3.3 frozendict

3.3.1 About

frozendict is an immutable wrapper around dictionaries that implements the complete mapping interface. It can be used as a drop-in replacement for dictionaries where immutability is desired.

Of course, this is python, and you can still poke around the object's internals if you want.

The *frozendict* constructor mimics dict, and all of the expected interfaces (iter, len, repr, hash, getitem) are provided. Note that a *frozendict* does not guarantee the immutability of its values, so the utility of the hash method is restricted by usage.

The only difference is that the copy() method of *frozendict* takes variable keyword arguments, which will be present as key/value pairs in the new, immutable copy.

3.3.2 Usage

```
>>> from cawdrey import frozendict
>>>
>>> fd = frozendict({ 'hello': 'World' })
>>>
>>> print fd
<frozendict {'hello': 'World'}>
>>>
>>> print fd['hello']
'World'
>>>
>>> print fd.copy(another='key/value')
<frozendict {'hello': 'World', 'another': 'key/value'}>
>>>
```

In addition, *frozendict* supports the + and - operands. If you add a *dict*-like object, a new *frozendict* will be returned, equal to the old *frozendict* updated with the other object. Example:

```
>>> frozendict({"Sulla": "Marco", 2: 3}) + {"Sulla": "Marò", 4: 7}
<frozendict {'Sulla': 'Marò', 2: 3, 4: 7}>
>>>
```

You can also subtract an iterable from a *frozendict*. A new *frozendict* will be returned, without the keys that are in the iterable. Examples:

```
>>> frozendict({"Sulla": "Marco", 2: 3}) - {"Sulla": "Marò", 4: 7}
<frozendict {'Sulla': 'Marco', 2: 3}>
>>> frozendict({"Sulla": "Marco", 2: 3}) - [2, 4]
<frozendict {'Sulla': 'Marco'}>
>>>
```

Some other examples:

```
>>> from cawdrey import frozendict
>>> fd = frozendict({"Sulla": "Marco", "Hicks": "Bill"})
>>> print(fd)
<frozendict {'Sulla': 'Marco', 'Hicks': 'Bill'}>
>>> print(fd["Sulla"])
Marco
>>> fd["Bim"]
KeyError: 'Bim'
>>> len(fd)
2
>>> "Sulla" in fd
True
>>> "Sulla" not in fd
False
>>> "Bim" in fd
False
>>> hash(fd)
835910019049608535
>>> fd_unhashable = frozendict({1: []})
>>> hash(fd_unhashable)
TypeError: unhashable type: 'list'
>>> fd2 = frozendict({"Hicks": "Bill", "Sulla": "Marco"})
>>> print(fd2)
<frozendict {'Hicks': 'Bill', 'Sulla': 'Marco'}>
>>> fd2 is fd
False
>>> fd2 == fd
True
>>> frozendict()
<frozendict {}>
>>> frozendict(Sulla="Marco", Hicks="Bill")
<frozendict {'Sulla': 'Marco', 'Hicks': 'Bill'}>
>>> frozendict((("Sulla", "Marco"), ("Hicks", "Bill")))
<frozendict {'Sulla': 'Marco', 'Hicks': 'Bill'}>
>>> fd.get("Sulla")
'Marco'
>>> print(fd.get("God"))
None
>>> tuple(fd.keys())
('Sulla', 'Hicks')
>>> tuple(fd.values())
('Marco', 'Bill')
>>> tuple(fd.items())
(('Sulla', 'Marco'), ('Hicks', 'Bill'))
>>> iter(fd)
<dict_keyiterator object at 0x7feb75c49188>
>>> frozendict.fromkeys(["Marco", "Giulia"], "Sulla")
<frozendict {'Marco': 'Sulla', 'Giulia': 'Sulla'}>
>>> fd["Sulla"] = "Silla"
TypeError: 'frozendict' object does not support item assignment
>>> del fd["Sulla"]
TypeError: 'frozendict' object does not support item deletion
>>> fd.clear()
AttributeError: 'frozendict' object has no attribute 'clear'
>>> fd.pop("Sulla")
AttributeError: 'frozendict' object has no attribute 'pop'
>>> fd.popitem()
```

(continues on next page)

(continued from previous page)

```
AttributeError: 'frozendict' object has no attribute 'popitem'
>>> fd.setdefault("Sulla")
AttributeError: 'frozendict' object has no attribute 'setdefault'
>>> fd.update({"Bim": "James May"})
AttributeError: 'frozendict' object has no attribute 'update'
```

3.3.3 API Reference

class cawdrey.**frozendict**(**args*, ***kwds*)

An immutable wrapper around dictionaries that implements the complete `collections.Mapping` interface. It can be used as a drop-in replacement for dictionaries where immutability is desired.

__abstractmethods__ = frozenset({})

__add__(*other*, **args*, ***kwargs*)

If you add a dict-like object, a new frozendict will be returned, equal to the old frozendict updated with the other object.

__and__(*other*, **args*, ***kwargs*)

Returns a new `frozendict`, that is the intersection between `self` and `other`.

If `other` is a *dict*-like object, the intersection will contain only the *items* in common.

If `other` is another iterable, the intersection will contain the items of `self` which keys are in *other*.

Iterables of pairs are *not* managed differently. This is for consistency.

Beware! The final order is dictated by the order of *other*. This allows the coder to change the order of the original `frozendict`.

The last two behaviours breaks voluntarily the `dict.items()` API, for consistency and practical reasons.

__annotations__ = {'dict_cls': typing.Union[typing.Type, NoneType]}

__args__ = None

__contains__(*key*)

Return type Any

__copy__(**args*, ***kwargs*)

__eq__(*other*)

Return self==value.

__extra__ = None

__getitem__(*key*)

Return type Any

__hash__()

Return hash(self).

Return type int

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

__iter__()

__len__()

Return type int

__module__ = 'cawdrey._frozendict'

static __new__(*cls*, **args*, ***kwds*)
Create and return a new object. See help(type) for accurate signature.

__next_in_mro__
alias of builtins.object

__orig_bases__ = (cawdrey.base.FrozenBase[~KT, ~VT],)

__origin__ = None

__parameters__ = (~KT, ~VT)

__repr__()
Return repr(self).

Return type str

__reversed__ = None

__slots__ = ()

__sub__(*other*, **args*, ***kwargs*)
The method will create a new frozendict, result of the subtraction by *other*.

If other is a dict-like, the result will have the items of the *frozendict* that are *not* in common with *other*.

If *other* is another type of iterable, the result will have the items of *frozendict* without the keys that are in *other*.

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__tree_hash__ = -9223366108873949231

__weakref__
list of weak references to the object (if defined)

copy(***add_or_replace*)

dict_cls
alias of builtins.dict

classmethod fromkeys(**args*, ***kwargs*)
Returns a new dict with keys from iterable and values equal to value.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

sorted(**args*, *by='keys'*, ***kwargs*)
Return a new *frozendict*, with the element insertion sorted. The signature is the same as the builtin sorted function, except for the additional parameter by, that is "keys" by default and can also be "values" and "items". So the resulting *frozendict* can be sorted by keys, values or items.

If you want more complicated sorts read the documentation of sorted.

The the parameters passed to the `key` function are the keys of the `frozendict` if `by = "keys"`, and are the items otherwise.

Note: Sorting by keys and items achieves the same effect. The only difference is when you want to customize the sorting passing a custom `key` function. You *could* achieve the same result using `by = "values"`, since also sorting by values passes the items to the key function. But this is an implementation detail and you should not rely on it.

---

values() → an object providing a view on D's values

3.3.4 Copyright

Based on https://github.com/slezica/python-frozendict and https://github.com/mredolatti/python-frozendict .
Copyright (c) 2012 Santiago Lezica
Licensed under the MIT License:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Also based on https://github.com/Marco-Sulla/python-frozendict
Copyright (c) Marco Sulla
Licensed under the GNU Lesser General Public License Version 3

3.4 FrozenOrderedDict

3.4.1 About

FrozenOrderedDict is a immutable wrapper around an OrderedDict.

FrozenOrderedDict is similar to *frozendict*, and with regards to immutability it solves the same problems:

- Because dictionaries are mutable, they are not hashable and cannot be used in sets or as dictionary keys.
- Nasty bugs can and do occur when mutable data structures are passed around.

It can be initialized just like a `dict` or `OrderedDict`. Once instantiated, an instance of *FrozenOrderedDict* cannot be altered, since it does not implement the `MutableMapping` interface.

It does implement the `Mapping` interface, so can be used just like a normal dictionary in most cases.

In order to modify the contents of a *FrozenOrderedDict*, a new instance must be created. The easiest way to do that is by calling the *.copy()* method. It will return a new instance of *FrozenOrderedDict* initialized using the following steps:

1. A copy of the wrapped OrderedDict instance will be created.
2. If any arguments or keyword arguments are passed to the *.copy()* method, they will be used to create another OrderedDict instance, which will then be used to update the copy made in step #1.
3. Finally, *self.__class__()* will be called, passing the copy as the only argument.

3.4.2 API Reference

class cawdrey.**FrozenOrderedDict**(**args*, ***kwds*)

An immutable OrderedDict. It can be used as a drop-in replacement for dictionaries where immutability is desired.

__abstractmethods__ = frozenset({})

__annotations__ = {'dict_cls': typing.Union[typing.Type, NoneType]}

__args__ = None

__contains__(*key*)

Return type Any

__copy__(**args*, ***kwargs*)

__eq__(*other*)

Return self==value.

__extra__ = None

__getitem__(*key*)

Return type Any

__hash__()

Return hash(self).

Return type int

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

__iter__()

__len__()

Return type int

__module__ = 'cawdrey.frozenordereddict'

static __new__(*cls*, **args*, ***kwds*)

Create and return a new object. See help(type) for accurate signature.

__next_in_mro__

alias of builtins.object

__orig_bases__ = (cawdrey.base.FrozenBase[~KT, ~VT],)

__origin__ = None

__parameters__ = (~KT, ~VT)

__repr__()
Return repr(self).

Return type str

__reversed__ = None

__slots__ = ()

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__tree_hash__ = -9223366108873949845

__weakref__
list of weak references to the object (if defined)

copy(**args*, ***kwargs*)
Return a copy of the *FrozenOrderedDict*.

Parameters

- **args** –
- **kwargs** –

Returns

Return type

dict_cls
alias of collections.OrderedDict

classmethod fromkeys(**args*, ***kwargs*)
Returns a new dict with keys from iterable and values equal to value.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

values() → an object providing a view on D's values

3.4.3 Copyright

Based on https://github.com/slezica/python-frozendict and https://github.com/mredolatti/python-frozendict .

3.5 NonelessDict

3.5.1 About

NonelessDict is a wrapper around dict that will check if a value is None/empty/False, and not add the key in that case.

The class has a method *set_with_strict_none_check()* that can be used to set a value and check only for None values.

NonelessOrderedDict is based *NonelessDict* and OrderedDict, so the order of key insertion is preserved.

3.5.2 Usage

3.5.3 API Reference

Provides frozendict, a simple immutable dictionary.

class cawdrey.nonelessdict.**NonelessDict**(**args*, ***kwds*)

A wrapper around dict that will check if a value is None/empty/False, and not add the key in that case. Use the set_with_strict_none_check function to check only for None

__abstractmethods__ = frozenset({})

__annotations__ = {'_dict': <class 'dict'>}

__args__ = None

__contains__(*key*)

Return type Any

__copy__(**args*, ***kwargs*)

__delitem__(*key*)

__eq__(*other*)

Return self==value.

__extra__ = None

__getitem__(*key*)

Return type Any

__hash__()

Return hash(self).

Return type int

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

__iter__()

__len__()

Return type int

__module__ = 'cawdrey.nonelessdict'

static __new__(*cls*, **args*, ***kwds*)

Create and return a new object. See help(type) for accurate signature.

__next_in_mro__

alias of builtins.object

__orig_bases__ = (cawdrey.base.MutableBase[~KT, ~VT],)

__origin__ = None

__parameters__ = (~KT, ~VT)

__repr__()

Return repr(self).

Return type str

__reversed__ = None

__setitem__(*key*, *value*)

__slots__ = ()

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__tree_hash__ = -9223366108873948868

__weakref__
list of weak references to the object (if defined)

clear() → None. Remove all items from D.

copy(***add_or_replace*)

dict_cls
alias of `builtins.dict`

classmethod fromkeys(**args*, ***kwargs*)
Returns a new dict with keys from iterable and values equal to value.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

set_with_strict_none_check(*key*, *value*)

Return type `None`

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

class `cawdrey.nonelessdict.`**NonelessOrderedDict**(**args*, ***kwds*)
A wrapper around OrderedDict that will check if a value is None/empty/False, and not add the key in that case. Use the set_with_strict_none_check function to check only for None

__abstractmethods__ = frozenset({})

__annotations__ = {'_dict': <class 'dict'>}

__args__ = None

__contains__(*key*)

Return type `Any`

__copy__(**args*, ***kwargs*)

__delitem__(*key*)

__eq__(*other*)
Return self==value.

__extra__ = None

__getitem__(*key*)

Return type Any

__hash__()
Return hash(self).

Return type int

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()

__len__()

Return type int

__module__ = 'cawdrey.nonelessdict'

static __new__(*cls*, **args*, ***kwds*)
Create and return a new object. See help(type) for accurate signature.

__next_in_mro__
alias of builtins.object

__orig_bases__ = (cawdrey.base.MutableBase[~KT, ~VT],)

__origin__ = None

__parameters__ = (~KT, ~VT)

__repr__()
Return repr(self).

Return type str

__reversed__ = None

__setitem__(*key*, *value*)

__slots__ = ()

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__tree_hash__ = -9223366108873948739

__weakref__
list of weak references to the object (if defined)

clear() → None. Remove all items from D.

copy(**args*, ***kwargs*)

dict_cls
alias of collections.OrderedDict

classmethod fromkeys(**args*, ***kwargs*)
Returns a new dict with keys from iterable and values equal to value.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

set_with_strict_none_check(*key*, *value*)

Return type None

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

3.5.4 Copyright

Based on https://github.com/slezica/python-frozendict and https://github.com/jerr0328/python-helpfuldicts .

3.6 Base Class

3.6.1 About

`FrozenBase` is the base class for *`frozendict`* and *`FrozenOrderedDict`*. If you wish to construct your own frozen dictionary classes, you may wish to inherit from this class.

3.6.2 Usage

3.6.3 API Reference

Base Classes

class cawdrey.base.**DictWrapper**(**args*, ***kwds*)

Abstract Mixin class for classes that wrap a dict object or similar

__abstractmethods__ = frozenset({'copy'})

__annotations__ = {'_dict': <class 'dict'>}

__args__ = None

__contains__(*key*)

Return type Any

__copy__(**args*, ***kwargs*)

__eq__(*other*)

Return self==value.

__extra__ = None

__getitem__(*key*)

Return type Any

__hash__ = None

__iter__()

__len__()

Return type int

__module__ = 'cawdrey.base'

static __new__(*cls*, **args*, ***kwds*)

Create and return a new object. See help(type) for accurate signature.

__next_in_mro__

alias of `builtins.object`

__orig_bases__ = (typing.Mapping[~KT, ~VT],)

__origin__ = None

__parameters__ = (~KT, ~VT)

__repr__()

Return repr(self).

Return type str

__reversed__ = None

__slots__ = ()

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__tree_hash__ = -9223366108873951365

__weakref__
list of weak references to the object (if defined)

_abc_cache = <_weakrefset.WeakSet object>

_abc_generic_negative_cache = <_weakrefset.WeakSet object>

_abc_generic_negative_cache_version = 42

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 42

_abc_registry = <_weakrefset.WeakSet object>

_dict: dict

_gorg
alias of *DictWrapper*

abstract copy(**args*, ***kwargs*)

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

values() → an object providing a view on D's values

class cawdrey.base.**FrozenBase**(**args*, ***kwds*)
Abstract Base Class for Frozen dictionaries

Used by frozendict and FrozenOrderedDict.

Custom subclasses must implement at a minimum `__init__`, `copy`, `fromkeys`.

__abstractmethods__ = frozenset({'__init__', 'copy'})

__annotations__ = {'dict_cls': typing.Union[typing.Type, NoneType]}

__args__ = None

__contains__(*key*)

Return type Any

__copy__(**args*, ***kwargs*)

__eq__(*other*)
Return self==value.

__extra__ = None

__getitem__(*key*)

Return type Any

__hash__ = None

abstract __init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()

__len__()

Return type int

__module__ = 'cawdrey.base'

static __new__(*cls*, **args*, ***kwds*)
Create and return a new object. See help(type) for accurate signature.

__next_in_mro__
alias of builtins.object

__orig_bases__ = (cawdrey.base.DictWrapper[~KT, ~VT],)

__origin__ = None

__parameters__ = (~KT, ~VT)

__repr__()
Return repr(self).

Return type str

__reversed__ = None

__slots__ = ()

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__tree_hash__ = -9223366108873951199

__weakref__
list of weak references to the object (if defined)

_abc_cache = <_weakrefset.WeakSet object>

_abc_generic_negative_cache = <_weakrefset.WeakSet object>

_abc_generic_negative_cache_version = 42

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 42

_abc_registry = <_weakrefset.WeakSet object>

_gorg
alias of *FrozenBase*

abstract copy(**args*, ***kwargs*)

dict_cls: Optional[Type] = None

classmethod fromkeys(**args*, ***kwargs*)
Returns a new dict with keys from iterable and values equal to value.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

values() → an object providing a view on D's values

class cawdrey.base.**MutableBase**(**args*, ***kwds*)
Abstract Base Class for mutable dictionaries

Used by NonelessDict and NonelessOrderedDict.

Custom subclasses must implement at a minimum `__init__`, `copy`, `fromkeys`.

_MutableMapping__marker = <object object>

__abstractmethods__ = frozenset({'__init__', 'copy'})

__annotations__ = {'_dict': <class 'dict'>}

__args__ = None

__contains__(*key*)

Return type Any

__copy__(**args*, ***kwargs*)

__delitem__(*key*)

__eq__(*other*)
Return self==value.

__extra__ = None

__getitem__(*key*)

Return type Any

__hash__ = None

abstract __init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()

__len__()

Return type int

__module__ = 'cawdrey.base'

static __new__(*cls*, **args*, ***kwds*)
Create and return a new object. See help(type) for accurate signature.

__next_in_mro__
alias of builtins.object

__orig_bases__ = (cawdrey.base.DictWrapper[~KT, ~VT], typing.MutableMapping[~KT, ~VT])

__origin__ = None

__parameters__ = (~KT, ~VT)

__repr__()
Return repr(self).

Return type str

__reversed__ = None

__setitem__(*key*, *value*)

__slots__ = ()

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__tree_hash__ = -9223366108873950070

__weakref__
list of weak references to the object (if defined)

_abc_cache = <_weakrefset.WeakSet object>

_abc_generic_negative_cache = <_weakrefset.WeakSet object>

_abc_generic_negative_cache_version = 42

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 42

_abc_registry = <_weakrefset.WeakSet object>

_dict

_gorg
alias of *MutableBase*

clear() → None. Remove all items from D.

abstract copy(**args*, ***kwargs*)

dict_cls = None

classmethod fromkeys(**args*, ***kwargs*)
Returns a new dict with keys from iterable and values equal to value.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

3.7 Functions

cawdrey.alphadict.**alphabetical_dict**(***kwargs*)

Returns an OrderedDict with the keys sorted alphabetically.

Parameters **kwargs** –

Returns

Return type

3.8 Contributing

Cawdrey uses tox to automate testing and packaging, and pre-commit to maintain code quality.

Install pre-commit with pip and install the git hook:

```
$ python -m pip install pre-commit
$ pre-commit install
```

3.8.1 Coding style

Yapf is used for code formatting, and isort is used to sort imports.

yapf and isort can be run manually via pre-commit:

```
$ pre-commit run yapf -a
$ pre-commit run isort -a
```

The complete autoformatting suite can be run with pre-commit:

```
$ pre-commit run -a
```

3.8.2 Automated tests

Tests are run with tox and pytest. To run tests for a specific Python version, such as Python 3.6, run:

```
$ tox -e py36
```

To run tests for all Python versions, simply run:

```
$ tox
```

3.8.3 Type Annotations

Type annotations are checked using mypy. Run mypy using tox:

```
$ tox -e mypy
```

3.8.4 Build documentation locally

The documentation is powered by Sphinx. A local copy of the documentation can be built with `tox`:

```
$ tox -e docs
```

3.9 Downloading source code

The `Cawdrey` source code is available on GitHub, and can be accessed from the following URL: https://github.com/domdfcoding/cawdrey"

If you have `git` installed, you can clone the repository with the following command:

```
$ git clone https://github.com/domdfcoding/cawdrey"
> Cloning into 'cawdrey'...
> remote: Enumerating objects: 47, done.
> remote: Counting objects: 100% (47/47), done.
> remote: Compressing objects: 100% (41/41), done.
> remote: Total 173 (delta 16), reused 17 (delta 6), pack-reused 126
> Receiving objects: 100% (173/173), 126.56 KiB | 678.00 KiB/s, done.
> Resolving deltas: 100% (66/66), done.
```

Alternatively, the code can be downloaded in a 'zip' file by clicking:
Clone or download –> Download Zip

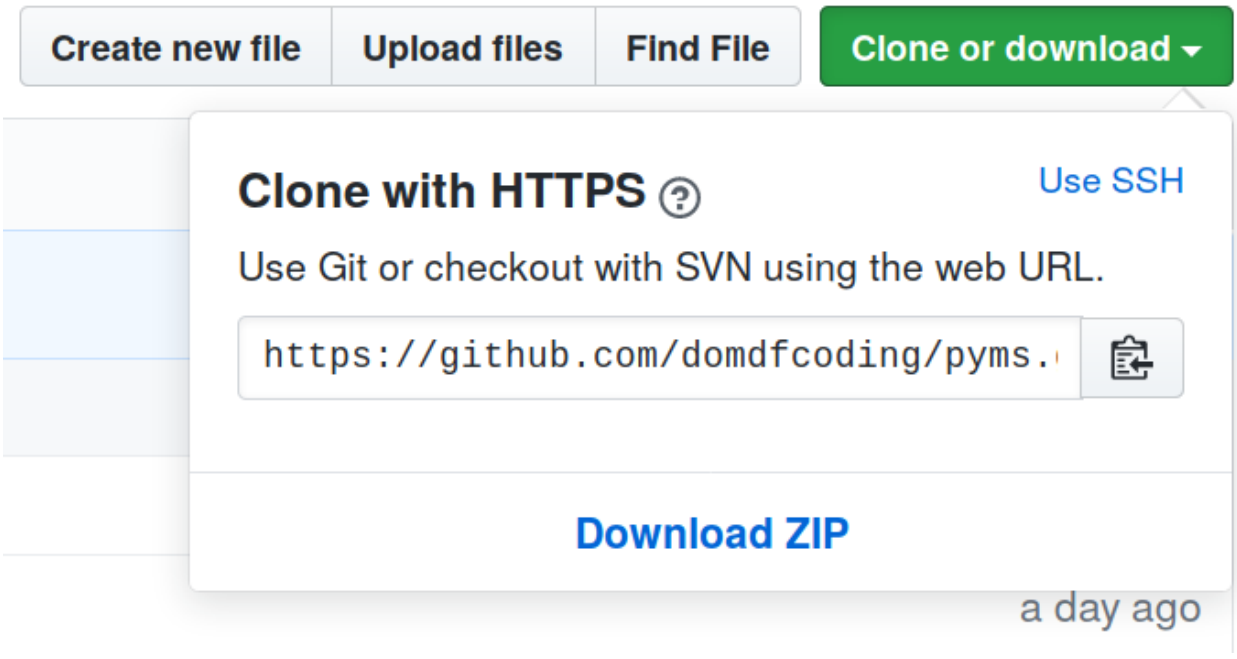

Fig. 1: Downloading a 'zip' file of the source code

3.9.1 Building from source

The recommended way to build `Cawdrey` is to use tox:

```
$ tox -e build
```

The source and wheel distributions will be in the directory `dist`.

If you wish, you may also use pep517.build or another **PEP 517**-compatible build tool.

View the Function Index or browse the Source Code.

Browse the GitHub Repository

CHAPTER

FOUR

AND FINALLY:

Why “Cawdrey”?

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

F

G

I

K

M

N

P

S

U

V