Cawdrey

Release 0.4.1

Dominic Davis-Foster

Mar 10, 2021

CONTENTS

Several useful custom dictionaries for Python

Docs	
Tests	
PyPI	
Anaconda	
Activity	
QA	
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.
- *Tally*: A subclass of `collections.Counter` with additional methods.
- *HeaderMapping*: A `collections.abc.MutableMapping` which supports duplicate, case-insentive keys.

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

from Anaconda

from GitHub

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

First add the required channels

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

Then install

```
$ conda install cawdrey
```

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

3.1 AlphaDict

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

Bases: *FrozenOrderedDict*[~KT, ~VT]

Initialize an immutable, alphabetised dictionary.

The signature is the same as regular dictionaries.

`AlphaDict()` -> new empty AlphaDict
`AlphaDict(mapping)` -> new AlphaDict initialized from a mapping object's (key, value) pairs
`AlphaDict(iterable)` -> new AlphaDict initialized as if via:

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

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

For example:

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

Methods:

`__contains__`(key)	Return `key in self`.
`__eq__`(other)	Return `self == other`.
`__getitem__`(key)	Return `self[key]`.
`__iter__`()	Iterates over the dictionary's keys.
`__len__`()	Returns the number of keys in the dictionary.
`__repr__`()	Return a string representation of the `DictWrapper`.
`copy`(*args, **kwargs)	Return a copy of the `FrozenOrderedDict`.
`fromkeys`(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
`get`(k[, default])	Return the value for `k` if `k` is in the dictionary, else `default`.
`items`()	Returns a set-like object providing a view on the `FrozenOrderedDict`'s items.
`keys`()	Returns a set-like object providing a view on the `FrozenOrderedDict`'s keys.
`values`()	Returns an object providing a view on the `FrozenOrderedDict`'s values.

__contains__(*key*)
Return `key in self`.

Parameters **key** (`object`)

Return type `bool`

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

Return type `bool`

__getitem__(*key*)
Return `self[key]`.

Parameters **key** (`~KT`)

Return type `~VT`

__iter__()
Iterates over the dictionary's keys.

Return type `Iterator[~KT]`

__len__()
Returns the number of keys in the dictionary.

Return type `int`

__repr__()
Return a string representation of the `DictWrapper`.

Return type `str`

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

Parameters

- **args**
- **kwargs**

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

Create a new dictionary with keys from iterable and values set to value.

Return type `FrozenBase[~KT, ~VT]`

get(*k*, *default=None*)

Return the value for `k` if `k` is in the dictionary, else `default`.

Parameters

- **k** – The key to return the value for.
- **default** – The value to return if `key` is not in the dictionary. Default `None`.

items()

Returns a set-like object providing a view on the `FrozenOrderedDict`'s items.

Return type `AbstractSet[Tuple[~KT, ~VT]]`

keys()

Returns a set-like object providing a view on the `FrozenOrderedDict`'s keys.

Return type `AbstractSet[~KT]`

values()

Returns an object providing a view on the `FrozenOrderedDict`'s values.

Return type `ValuesView[~VT]`

3.2 bdict

class bdict(*seq=None*, ***kwargs*)

Bases: `UserDict`

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

Methods:

`__contains__`(key)	Return `key in self`.
`__delitem__`(key)	Delete `self[key]`.
`__eq__`(other)	Return `self == other`.
`__getitem__`(key)	Return `self[key]`.
`__repr__`()	Return repr(self).
`__setitem__`(key, val)	Set `self[key]` to `value`.
`clear`()	Removes all items from the `bdict`.
`get`(k[, default])	Return the value for `k` if `k` is in the dictionary, else `default`.
`items`()	Returns a set-like object providing a view on the `bdict`'s items.
`keys`()	Returns a set-like object providing a view on the `bdict`'s keys.
`pop`(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
`popitem`()	as a 2-tuple; but raise KeyError if D is empty.
`setdefault`(k[,d])	
`update`([E,]**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`()	Returns an object providing a view on the `bdict`'s values.

__contains__(*key*)

Return `key in self`.

Parameters **key** (`object`)

Return type `bool`

__delitem__(*key*)

Delete `self[key]`.

Parameters **key** (`~KT`)

__eq__(*other*)

Return `self == other`.

Return type `bool`

__getitem__(*key*)

Return `self[key]`.

Parameters **key** (`~KT`)

Return type `~VT`

__repr__()

Return repr(self).

__setitem__(*key*, *val*)

Set `self[key]` to `value`.

Parameters

- **key**
- **val**

clear()
Removes all items from the *bdict*.

get(*k*, *default=None*)
Return the value for `k` if `k` is in the dictionary, else `default`.

Parameters

- **k** – The key to return the value for.
- **default** – The value to return if `key` is not in the dictionary. Default `None`.

Overloads

- *get*(**k**: `~KT`) -> `Optional`[`~VT`]
- *get*(**k**: `~KT`, **default**: `Union`[`~VT`, `~T`]) -> `Union`[`~VT`, `~T`]

items()
Returns a set-like object providing a view on the *bdict*'s items.

Return type `AbstractSet`[`Tuple`[`~KT`, `~VT`]]

keys()
Returns a set-like object providing a view on the *bdict*'s keys.

Return type `AbstractSet`[`~KT`]

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()
Returns an object providing a view on the *bdict*'s values.

Return type `ValuesView`[`~VT`]

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)
```

(continues on next page)

(continued from previous page)

```
<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()
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 frozendict(**args*, ***kwargs*)

Bases: *FrozenBase*[~KT, ~VT]

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.

Methods:

__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.

continues on next page

Table 3 – continued from previous page

`__and__`(other, *args, **kwargs)	Returns a new `frozendict`, that is the intersection between `self` and `other`.
`__contains__`(key)	Return `key in self`.
`__eq__`(other)	Return `self == other`.
`__getitem__`(key)	Return `self[key]`.
`__iter__`()	Iterates over the dictionary's keys.
`__len__`()	Returns the number of keys in the dictionary.
`__repr__`()	Return a string representation of the `DictWrapper`.
`__sub__`(other, *args, **kwargs)	The method will create a new `frozendict`, result of the subtraction by *other*.
`copy`(*args, **kwargs)	Return a copy of the dictionary.
`fromkeys`(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
`get`(k[, default])	Return the value for `k` if `k` is in the dictionary, else `default`.
`items`()	Returns a set-like object providing a view on the `bdict`'s items.
`keys`()	Returns a set-like object providing a view on the `bdict`'s keys.
`sorted`(*args[, by])	Return a new `frozendict`, with the element insertion sorted.
`values`()	Returns an object providing a view on the `bdict`'s values.

__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.

__contains__(*key*)

Return `key in self`.

Parameters **key** (`object`)

Return type `bool`

__eq__(*other*)

Return `self == other`.

Return type `bool`

__getitem__(*key*)

Return `self[key]`.

Parameters **key** (~KT)

Return type ~VT

__iter__()

Iterates over the dictionary's keys.

Return type Iterator[~KT]

__len__()

Returns the number of keys in the dictionary.

Return type int

__repr__()

Return a string representation of the *DictWrapper*.

Return type str

__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.

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

Return a copy of the dictionary.

Return type ~_D

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

Create a new dictionary with keys from iterable and values set to value.

Return type *FrozenBase*[~KT, ~VT]

get(*k*, *default=None*)

Return the value for k if k is in the dictionary, else default.

Parameters

- **k** – The key to return the value for.
- **default** – The value to return if key is not in the dictionary. Default None.

items()

Returns a set-like object providing a view on the *bdict*'s items.

Return type AbstractSet[Tuple[~KT, ~VT]]

keys()

Returns a set-like object providing a view on the *bdict*'s keys.

Return type AbstractSet[~KT]

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()

Returns an object providing a view on the *bdict*'s values.

Return type `ValuesView[~VT]`

3.3.4 Copyright

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

Also based on https://github.com/Marco-Sulla/python-frozendict

3.4 FrozenOrderedDict

3.4.1 About

FrozenOrderedDict is a immutable wrapper around an `OrderedDict`. It 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.

`FrozenOrderedDict` implements 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 FrozenOrderedDict(**args*, ***kwargs*)

Bases: `FrozenBase`[~KT, ~VT]

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

Methods:

`__contains__`(key)	Return `key in self`.
`__eq__`(other)	Return `self == other`.
`__getitem__`(key)	Return `self[key]`.
`__iter__`()	Iterates over the dictionary's keys.
`__len__`()	Returns the number of keys in the dictionary.
`__repr__`()	Return a string representation of the `DictWrapper`.
`copy`(*args, **kwargs)	Return a copy of the `FrozenOrderedDict`.
`fromkeys`(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
`get`(k[, default])	Return the value for `k` if `k` is in the dictionary, else `default`.
`items`()	Returns a set-like object providing a view on the `FrozenOrderedDict`'s items.
`keys`()	Returns a set-like object providing a view on the `FrozenOrderedDict`'s keys.
`values`()	Returns an object providing a view on the `FrozenOrderedDict`'s values.

__contains__(*key*)

Return `key in self`.

Parameters **key** (`object`)

Return type `bool`

__eq__(*other*)

Return `self == other`.

Return type `bool`

__getitem__(*key*)

Return `self[key]`.

Parameters **key** (~KT)

Return type ~VT

__iter__()

Iterates over the dictionary's keys.

Return type Iterator[~KT]

__len__()

Returns the number of keys in the dictionary.

Return type int

__repr__()

Return a string representation of the *DictWrapper*.

Return type str

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

Return a copy of the *FrozenOrderedDict*.

Parameters

- **args**
- **kwargs**

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

Create a new dictionary with keys from iterable and values set to value.

Return type *FrozenBase*[~KT, ~VT]

get(*k*, *default=None*)

Return the value for k if k is in the dictionary, else default.

Parameters

- **k** – The key to return the value for.
- **default** – The value to return if key is not in the dictionary. Default None.

Overloads

- *get*(**k**: ~KT) -> Optional[~VT]
- *get*(**k**: ~KT, **default**: Union[~VT, ~T]) -> Union[~VT, ~T]

items()

Returns a set-like object providing a view on the *FrozenOrderedDict*'s items.

Return type AbstractSet[Tuple[~KT, ~VT]]

keys()

Returns a set-like object providing a view on the *FrozenOrderedDict*'s keys.

Return type AbstractSet[~KT]

values()

Returns an object providing a view on the *FrozenOrderedDict*'s values.

Return type ValuesView[~VT]

3.4.3 Copyright

3.5 HeaderMapping

`collections.abc.MutableMapping` which supports duplicate, case-insentive keys.

New in version 0.4.0.

Classes:

HeaderMapping()	Provides a `MutableMapping` interface to a list of headers, such as those used in an email message.

class HeaderMapping

Bases: `MutableMapping[str, ~VT]`

Provides a `MutableMapping` interface to a list of headers, such as those used in an email message.

See also:

`email.message.Message` and `email.message.EmailMessage`

`MutableMapping` interface, which assumes there is exactly one occurrence of the header per mapping. Some headers do in fact appear multiple times, and for those headers you must use the *get_all()* method to obtain all values for that key.

Methods:

__contains__(name)	Returns whether `name` is in the *HeaderMapping*.
__delitem__(name)	Delete all occurrences of a header, if present.
__getitem__(name)	Get a header value.
__iter__()	Returns an iterator over the keys in the *HeaderMapping*.
__len__()	Return the total number of keys, including duplicates.
__repr__()	Return a string representation of the *HeaderMapping*.
__setitem__(name, val)	Set the value of a header.
get(k[, default])	Get a header value.
get_all(k[, default])	Return a list of all the values for the named field.
items()	Get all the message's header fields and values.
keys()	Return a list of all the message's header field names.
values()	Return a list of all the message's header values.

__contains__(*name*)

Returns whether `name` is in the *HeaderMapping*.

Parameters **name** (`object`)

Return type `bool`

__delitem__(*name*)

Delete all occurrences of a header, if present.

Does not raise an exception if the header is missing.

Parameters **name** (`str`)

__getitem__(*name*)

Get a header value.

Note: If the header appears multiple times, exactly which occurrence gets returned is undefined. Use the `get_all()` method to get all values matching a header field name.

Parameters **name** (`str`)

Return type `~VT`

__iter__()

Returns an iterator over the keys in the `HeaderMapping`.

Return type `Iterator[str]`

__len__()

Return the total number of keys, including duplicates.

Return type `int`

__repr__()

Return a string representation of the `HeaderMapping`.

New in version 0.4.1.

Return type `str`

__setitem__(*name*, *val*)

Set the value of a header.

Parameters

- **name** (`str`)
- **val** (`~VT`)

get(*k*, *default=None*)

Get a header value.

Like `__getitem__()`, but returns `default` instead of `None` when the field is missing.

Parameters

- **k** (`str`)
- **default** – Default `None`.

Overloads

- *get*(**k**: `str`) -> `Optional[~VT]`
- *get*(**k**: `str`, **default**: `Union[~VT, ~_T]`) -> `Union[~VT, ~_T]`

get_all(*k*, *default=None*)

Return a list of all the values for the named field.

These will be sorted in the order they appeared in the original message, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list.

If no such fields exist, `default` is returned.

Parameters

- **k** (`str`)
- **default** – Default `None`.

Overloads

- *get_all*(**k**: str) -> Optional[List[~VT]]
- *get_all*(**k**: str, **default**: Union[~VT, ~_T]) -> Union[List[~VT], ~_T]

items()

Get all the message's header fields and values.

These will be sorted in the order they appeared in the original message, or were added to the message, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list.

Return type List[Tuple[str, ~VT]]

keys()

Return a list of all the message's header field names.

These will be sorted in the order they appeared in the original message, or were added to the message, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list.

Return type List[str]

values()

Return a list of all the message's header values.

These will be sorted in the order they appeared in the original message, or were added to the message, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list.

Return type List[~VT]

3.6 NonelessDict

3.6.1 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 on *NonelessDict* and OrderedDict, so the order of key insertion is preserved.

3.6.2 API Reference

Provides NonelessDict.

Classes:

NonelessDict(*args, **kwargs)	A wrapper around dict that will check if a value is None/empty/False, and not add the key in that case.
NonelessOrderedDict(*args, **kwargs)	A wrapper around OrderedDict that will check if a value is None/empty/False, and not add the key in that case.

class NonelessDict(**args*, ***kwargs*)

Bases: *MutableBase*[~KT, ~VT]

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()* method to check only for None.

Methods:

`__contains__`(key)	Return `key in self`.
`__delitem__`(key)	Delete `self[key]`.
`__eq__`(other)	Return `self == other`.
`__getitem__`(key)	Return `self[key]`.
`__iter__`()	Iterates over the dictionary's keys.
`__len__`()	Returns the number of keys in the dictionary.
`__repr__`()	Return a string representation of the `DictWrapper`.
`__setitem__`(key, value)	Set `self[key]` to `value`.
`clear`()	
`copy`(**add_or_replace)	Return a copy of the dictionary.
`fromkeys`(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
`get`(k[, default])	Return the value for `k` if `k` is in the dictionary, else `default`.
`items`()	Returns a set-like object providing a view on the `bdict`'s items.
`keys`()	Returns a set-like object providing a view on the `bdict`'s keys.
`pop`(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
`popitem`()	as a 2-tuple; but raise KeyError if D is empty.
`set_with_strict_none_check`(key, value)	Set `key` in the dictionary to `value`, but skipping `None` values.
`setdefault`(k[,d])	
`update`([E,]**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`()	Returns an object providing a view on the `bdict`'s values.

__contains__(*key*)

Return `key in self`.

Parameters **key** (`object`)

Return type `bool`

__delitem__(*key*)

Delete `self[key]`.

__eq__(*other*)

Return `self == other`.

Return type `bool`

__getitem__(*key*)

Return `self[key]`.

Parameters **key** (`~KT`)

Return type `~VT`

__iter__()
Iterates over the dictionary's keys.

Return type `Iterator[~KT]`

__len__()
Returns the number of keys in the dictionary.

Return type `int`

__repr__()
Return a string representation of the `DictWrapper`.

Return type `str`

__setitem__(*key*, *value*)
Set `self[key]` to `value`.

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

copy(***add_or_replace*)
Return a copy of the dictionary.

classmethod fromkeys(*iterable*, *value=None*)
Create a new dictionary with keys from iterable and values set to value.

Return type `MutableBase[~KT, ~VT]`

get(*k*, *default=None*)
Return the value for `k` if `k` is in the dictionary, else `default`.

Parameters

- **k** – The key to return the value for.
- **default** – The value to return if `key` is not in the dictionary. Default `None`.

items()
Returns a set-like object providing a view on the `bdict`'s items.

Return type `AbstractSet[Tuple[~KT, ~VT]]`

keys()
Returns a set-like object providing a view on the `bdict`'s keys.

Return type `AbstractSet[~KT]`

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*)
Set `key` in the dictionary to `value`, but skipping `None` values.

Parameters

- **key** (`~KT`)
- **value** (`Optional[~VT]`)

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()
Returns an object providing a view on the *bdict*'s values.

Return type ValuesView[~VT]

class NonelessOrderedDict(**args*, ***kwargs*)
Bases: *MutableBase*[~KT, ~VT]

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

Methods:

__contains__(key)	Return `key in self`.
__delitem__(key)	Delete `self[key]`.
__eq__(other)	Return `self == other`.
__getitem__(key)	Return `self[key]`.
__iter__()	Iterates over the dictionary's keys.
__len__()	Returns the number of keys in the dictionary.
__repr__()	Return a string representation of the *DictWrapper*.
__setitem__(key, value)	Set `self[key]` to `value`.
clear()	
copy(*args, **kwargs)	Return a copy of the dictionary.
fromkeys(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
get(k[, default])	Return the value for `k` if `k` is in the dictionary, else `default`.
items()	Returns a set-like object providing a view on the *bdict*'s items.
keys()	Returns a set-like object providing a view on the *bdict*'s keys.
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
set_with_strict_none_check(key, value)	Set `key` in the dictionary to `value`, but skipping `None` values.
setdefault(k[,d])	
update([E,]**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()	Returns an object providing a view on the *bdict*'s values.

__contains__(*key*)
Return `key in self`.

Parameters **key** (object)

Return type bool

__delitem__(*key*)
Delete `self[key]`.

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

Return type `bool`

__getitem__(*key*)
Return `self[key]`.

Parameters **key** (~KT)

Return type ~VT

__iter__()
Iterates over the dictionary's keys.

Return type `Iterator`[~KT]

__len__()
Returns the number of keys in the dictionary.

Return type `int`

__repr__()
Return a string representation of the `DictWrapper`.

Return type `str`

__setitem__(*key*, *value*)
Set `self[key]` to `value`.

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

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

classmethod fromkeys(*iterable*, *value=None*)
Create a new dictionary with keys from iterable and values set to value.

Return type `MutableBase`[~KT, ~VT]

get(*k*, *default=None*)
Return the value for `k` if `k` is in the dictionary, else `default`.

Parameters

- **k** – The key to return the value for.
- **default** – The value to return if `key` is not in the dictionary. Default `None`.

items()
Returns a set-like object providing a view on the `bdict`'s items.

Return type `AbstractSet`[`Tuple`[~KT, ~VT]]

keys()
Returns a set-like object providing a view on the `bdict`'s keys.

Return type `AbstractSet`[~KT]

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*)
Set `key` in the dictionary to `value`, but skipping `None` values.

Parameters

- **key** (~KT)
- **value** (Optional[~VT])

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()
Returns an object providing a view on the *bdict*'s values.

Return type ValuesView[~VT]

3.6.3 Copyright

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

3.7 Tally

Subclass of collections.Counter with additional methods.

New in version 0.3.0.

Data:

_F	Invariant TypeVar constrained to float, int and numbers.Real.

Classes:

SupportsMostCommon	typing.Protocol for classes which support a collections.Counter-like collections.Counter.most_common() method.

continues on next page

Table 11 – continued from previous page

Tally([iterable])	Subclass of `collections.Counter` with additional methods.
Percentage	Provides a dictionary interface, but with `collections.Counter`'s `collections.Counter.most_common()` method.

_F = TypeVar(_F, float, int, Real)

Type: `TypeVar`

Invariant `TypeVar` constrained to `float`, `int` and `numbers.Real`.

protocol SupportsMostCommon

Bases: `typing.Protocol`

`typing.Protocol` for classes which support a `collections.Counter`-like `collections.Counter.most_common()` method.

This protocol is runtime checkable.

Classes that implement this protocol must have the following methods / attributes:

items()

Returns an iterator over the mapping's items (as `(key, value)` pairs).

Return type `Iterable[Tuple[~KT, float]]`

most_common(*n=None*)

List the `n` most common elements and their counts from the most common to the least. If `n` is `None` then list all element counts.

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

Parameters **n** (`Optional[int]`) – Default `None`.

Return type `Union[List[Tuple[~KT, float]], List[Tuple[~KT, int]]]`

class Tally(*iterable=None*, */*, ***kwds*)

Bases: `Counter[~KT]`

Subclass of `collections.Counter` with additional methods.

New in version 0.3.0.

Methods:

as_percentage()	Returns the *Tally* as a `collections.OrderedDict` comprising the count for each element as a percentage of the sum of all elements.
get_percentage(item[, default])	Returns the count for `item`, as a percentage of the sum of all elements.
most_common([n])	List the `n` most common elements and their counts from the most common to the least.

Attributes:

total	Returns the total count for all elements.

as_percentage()

Returns the *Tally* as a collections.OrderedDict comprising the count for each element as a percentage of the sum of all elements.

Important: The sum of the dictionary's values may not add up to exactly 1.0 due to limitations of floating-point numbers.

Return type *Percentage*[~KT]

property total

Returns the total count for all elements.

Return type int

get_percentage(*item*, *default=None*)

Returns the count for item, as a percentage of the sum of all elements.

Parameters

- **item** (~KT)
- **default** (Optional[~_F]) – A default percentage (as a float) to return if item is not in the dictionary. Default None.

Return type Union[None, ~_F, float]

Overloads

- *get_percentage*(**item**: ~KT) -> Optional[float]
- *get_percentage*(**item**: ~KT, **default**: ~_F) -> Union[~_F, float]

most_common(*n=None*)

List the n most common elements and their counts from the most common to the least. If n is None then list all element counts.

```
>>> Tally('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

Parameters **n** (Optional[int]) – Default None.

Return type List[Tuple[~KT, int]]

class Percentage

Bases: Dict[~KT, float]

Provides a dictionary interface, but with collections.Counter's collections.Counter.most_common() method.

Represents the return type of *cawdrey.tally.Tally.as_percentage()*.

Methods:

__contains__(key, /)	True if the dictionary has the specified key, else False.

continues on next page

Table 14 – continued from previous page

__delattr__(name, /)	Implement delattr(self, name).
__delitem__(key, /)	Delete self[key].
__eq__(value, /)	Return self==value.
__format__(format_spec, /)	Default object formatter.
__ge__(value, /)	Return self>=value.
__getattribute__(name, /)	Return getattr(self, name).
__getitem__	x.__getitem__(y) <==> x[y]
__gt__(value, /)	Return self>value.
__iter__()	Implement iter(self).
__le__(value, /)	Return self<=value.
__len__()	Return len(self).
__lt__(value, /)	Return self<value.
__ne__(value, /)	Return self!=value.
__reduce__()	Helper for pickle.
__reduce_ex__(protocol, /)	Helper for pickle.
__repr__()	Return repr(self).
__reversed__()	Return a reverse iterator over the dict keys.
__setattr__(name, value, /)	Implement setattr(self, name, value).
__setitem__(key, value, /)	Set self[key] to value.
__sizeof__()	
__str__()	Return str(self).
clear()	
copy()	
fromkeys([value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	
most_common([n])	List the n most common elements and their counts from the most common to the least.

__contains__(*key*, /)

True if the dictionary has the specified key, else False.

__delattr__(*name*, /)

Implement delattr(self, name).

__delitem__(*key*, /)

Delete self[key].

__eq__(*value*, /)

Return self==value.

__format__(*format_spec*, /)
Default object formatter.

__ge__(*value*, /)
Return self>=value.

__getattribute__(*name*, /)
Return getattr(self, name).

__getitem__()
x.__getitem__(y) <==> x[y]

__gt__(*value*, /)
Return self>value.

__iter__()
Implement iter(self).

__le__(*value*, /)
Return self<=value.

__len__()
Return len(self).

__lt__(*value*, /)
Return self<value.

__ne__(*value*, /)
Return self!=value.

__reduce__()
Helper for pickle.

__reduce_ex__(*protocol*, /)
Helper for pickle.

__repr__()
Return repr(self).

__reversed__()
Return a reverse iterator over the dict keys.

__setattr__(*name*, *value*, /)
Implement setattr(self, name, value).

__setitem__(*key*, *value*, /)
Set self[key] to value.

__sizeof__() → size of D in memory, in bytes

__str__()
Return str(self).

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

copy() → a shallow copy of D

fromkeys(*value=None*, /)
Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, /)
Return the value for key if key is in the dictionary, else default.

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()
Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, /)
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

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

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

most_common(*n=None*)
List the `n` most common elements and their counts from the most common to the least. If `n` is `None` then list all element counts.

```
>>> Tally('abracadabra').as_percentage().most_common(3)
[('a', 0.45454545454545453), ('b', 0.18181818181818182), ('r', 0.
↪18181818181818182)]
```

Parameters **n** (`Optional[int]`) – Default `None`.

Return type `List[Tuple[~KT, float]]`

3.8 Base Classes

3.8.1 About

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

3.8.2 API Reference

Base Classes.

Classes:

`DictWrapper`(*args, **kwds)	Abstract Mixin class for classes that wrap a dict object or similar.
`FrozenBase`(*args, **kwargs)	Abstract Base Class for Frozen dictionaries.
`MutableBase`(*args, **kwargs)	Abstract Base Class for mutable dictionaries.

class DictWrapper(**args*, ***kwds*)
Bases: `Mapping[~KT, ~VT]`

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

Methods:

`__contains__`(key)	Return `key in self`.
`__eq__`(other)	Return `self == other`.
`__getitem__`(key)	Return `self[key]`.
`__iter__`()	Iterates over the dictionary's keys.
`__len__`()	Returns the number of keys in the dictionary.
`__repr__`()	Return a string representation of the `DictWrapper`.
`copy`(*args, **kwargs)	Return a copy of the dictionary.
`get`(k[, default])	Return the value for `k` if `k` is in the dictionary, else `default`.
`items`()	Returns a set-like object providing a view on the `bdict`'s items.
`keys`()	Returns a set-like object providing a view on the `bdict`'s keys.
`values`()	Returns an object providing a view on the `bdict`'s values.

__contains__(*key*)

Return `key in self`.

Parameters **key** (`object`)

Return type `bool`

__eq__(*other*)

Return `self == other`.

Return type `bool`

__getitem__(*key*)

Return `self[key]`.

Parameters **key** (`~KT`)

Return type `~VT`

__iter__()

Iterates over the dictionary's keys.

Return type `Iterator[~KT]`

__len__()

Returns the number of keys in the dictionary.

Return type `int`

__repr__()

Return a string representation of the `DictWrapper`.

Return type `str`

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

Return a copy of the dictionary.

Return type `~_D`

get(*k*, *default=None*)

Return the value for `k` if `k` is in the dictionary, else `default`.

Parameters

- **k** – The key to return the value for.
- **default** – The value to return if `key` is not in the dictionary. Default `None`.

Overloads

- *get*(**k**: ~KT) -> `Optional`[~VT]
- *get*(**k**: ~KT, **default**: `Union`[~VT, ~T]) -> `Union`[~VT, ~T]

items()

Returns a set-like object providing a view on the *bdict*'s items.

Return type `AbstractSet`[`Tuple`[~KT, ~VT]]

keys()

Returns a set-like object providing a view on the *bdict*'s keys.

Return type `AbstractSet`[~KT]

values()

Returns an object providing a view on the *bdict*'s values.

Return type `ValuesView`[~VT]

class FrozenBase(**args*, ***kwargs*)

Bases: *DictWrapper*[~KT, ~VT]

Abstract Base Class for Frozen dictionaries.

Used by *frozendict* and *FrozenOrderedDict*.

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

Methods:

__contains__(key)	Return `key in self`.
__eq__(other)	Return `self == other`.
__getitem__(key)	Return `self[key]`.
__iter__()	Iterates over the dictionary's keys.
__len__()	Returns the number of keys in the dictionary.
__repr__()	Return a string representation of the *DictWrapper*.
copy(*args, **kwargs)	Return a copy of the dictionary.
fromkeys(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
get(k[, default])	Return the value for `k` if `k` is in the dictionary, else `default`.
items()	Returns a set-like object providing a view on the *bdict*'s items.
keys()	Returns a set-like object providing a view on the *bdict*'s keys.
values()	Returns an object providing a view on the *bdict*'s values.

__contains__(*key*)

Return `key in self`.

Parameters **key** (`object`)

Return type `bool`

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

Return type `bool`

__getitem__(*key*)
Return `self[key]`.

Parameters **key** (`~KT`)

Return type `~VT`

__iter__()
Iterates over the dictionary's keys.

Return type `Iterator[~KT]`

__len__()
Returns the number of keys in the dictionary.

Return type `int`

__repr__()
Return a string representation of the *DictWrapper*.

Return type `str`

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

Return type `~_D`

classmethod fromkeys(*iterable*, *value=None*)
Create a new dictionary with keys from iterable and values set to value.

Return type *FrozenBase*[`~KT`, `~VT`]

Overloads

- *fromkeys*(**iterable**) -> `FrozenBase[KT, Any]`
- *fromkeys*(**iterable**, **value**) -> `FrozenBase[KT, VT]`

get(*k*, *default=None*)
Return the value for `k` if `k` is in the dictionary, else `default`.

Parameters

- **k** – The key to return the value for.
- **default** – The value to return if `key` is not in the dictionary. Default `None`.

items()
Returns a set-like object providing a view on the *bdict*'s items.

Return type `AbstractSet[Tuple[~KT, ~VT]]`

keys()
Returns a set-like object providing a view on the *bdict*'s keys.

Return type `AbstractSet[~KT]`

values()
Returns an object providing a view on the *bdict*'s values.

Return type ValuesView[~VT]

class MutableBase(**args*, ***kwargs*)
Bases: *DictWrapper*[~KT, ~VT], MutableMapping[~KT, ~VT]

Abstract Base Class for mutable dictionaries.

Used by *NonelessDict* and *NonelessOrderedDict*.

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

Methods:

__contains__(key)	Return key in self.
__delitem__(key)	Delete self[key].
__eq__(other)	Return self == other.
__getitem__(key)	Return self[key].
__iter__()	Iterates over the dictionary's keys.
__len__()	Returns the number of keys in the dictionary.
__repr__()	Return a string representation of the *DictWrapper*.
__setitem__(key, value)	Set self[key] to value.
clear()	
copy(*args, **kwargs)	Return a copy of the dictionary.
fromkeys(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
get(k[, default])	Return the value for k if k is in the dictionary, else default.
items()	Returns a set-like object providing a view on the *bdict*'s items.
keys()	Returns a set-like object providing a view on the *bdict*'s keys.
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
setdefault(k[,d])	
update([E,]**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()	Returns an object providing a view on the *bdict*'s values.

__contains__(*key*)
Return key in self.

Parameters **key** (object)

Return type bool

__delitem__(*key*)
Delete self[key].

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

Return type `bool`

__getitem__(*key*)
Return `self[key]`.

Parameters **key** (`~KT`)

Return type `~VT`

__iter__()
Iterates over the dictionary's keys.

Return type `Iterator[~KT]`

__len__()
Returns the number of keys in the dictionary.

Return type `int`

__repr__()
Return a string representation of the `DictWrapper`.

Return type `str`

__setitem__(*key*, *value*)
Set `self[key]` to `value`.

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

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

Return type `~_D`

classmethod fromkeys(*iterable*, *value=None*)
Create a new dictionary with keys from iterable and values set to value.

Return type `MutableBase[~KT, ~VT]`

Overloads

- `fromkeys`(**iterable**) -> `MutableBase[KT, Any]`
- `fromkeys`(**iterable**, **value**) -> `MutableBase[KT, VT]`

get(*k*, *default=None*)
Return the value for `k` if `k` is in the dictionary, else `default`.

Parameters

- **k** – The key to return the value for.
- **default** – The value to return if `key` is not in the dictionary. Default `None`.

items()
Returns a set-like object providing a view on the `bdict`'s items.

Return type `AbstractSet[Tuple[~KT, ~VT]]`

keys()
Returns a set-like object providing a view on the `bdict`'s keys.

Return type `AbstractSet[~KT]`

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()
Returns an object providing a view on the *bdict*'s values.

Return type ValuesView[~VT]

3.9 Functions

alphabetical_dict(**kwargs)
Returns an OrderedDict with the keys sorted alphabetically.

Parameters **kwargs**

Returns

Return type

search_dict(*dictionary*, *regex*)
Return the subset of the dictionary whose keys match the regex.

Parameters

- **dictionary** (Dict[str, Any])
- **regex** (Union[str, Pattern])

Return type Dict[str, Any]

3.10 Overview

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
```

3.11 Coding style

formate is used for code formatting.

It can be run manually via pre-commit:

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

Or, to run the complete autoformatting suite:

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

3.12 Automated tests

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

```
$ tox -e py36
```

To run tests for all Python versions, simply run:

```
$ tox
```

3.13 Type Annotations

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

```
$ tox -e mypy
```

3.14 Build documentation locally

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

```
$ tox -e docs
```

3.15 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

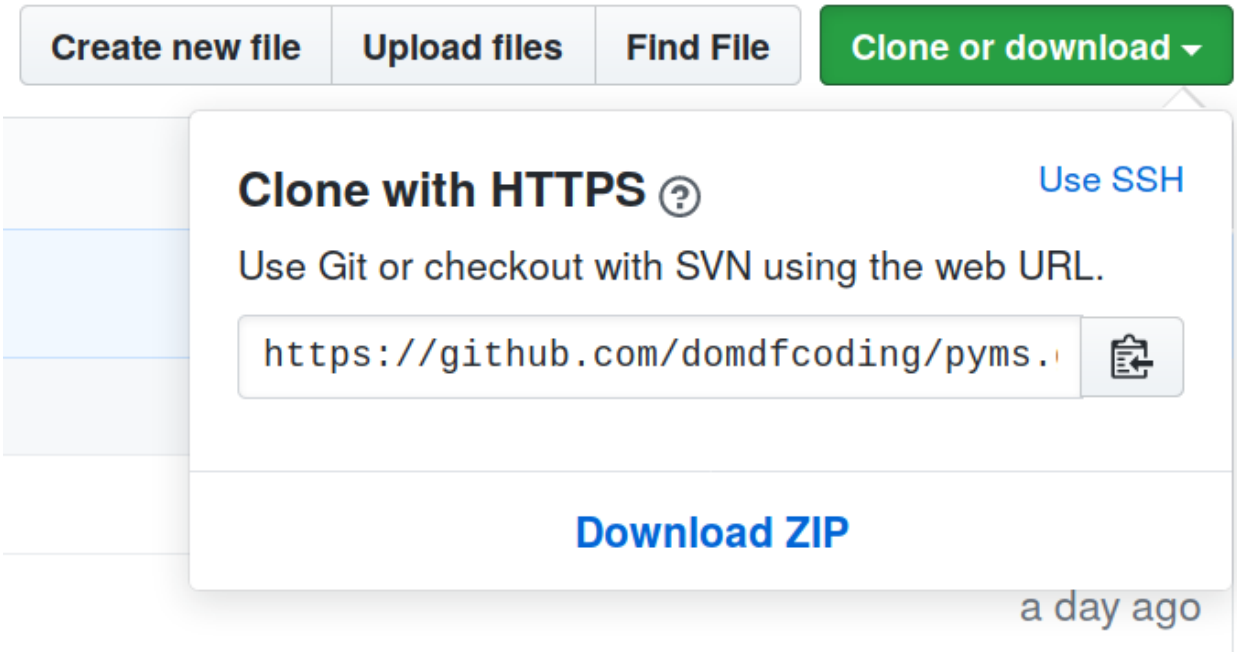

Fig. 1: Downloading a 'zip' file of the source code

3.15.1 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

H

I

K

M

N

P

S

T

U

V