UBelt Documentation

Release 1.1.0

Jon Crall

Jun 04, 2022

PACKAGE LAYOUT

UBelt is a "utility belt" of commonly needed utility and helper functions. It is a currated collection of top-level utilities with functionality that falls into a mixture of categories.

The source code is available at https://github.com/Erotemic/ubelt. We also have Jupyter notebook demos.

The ubelt API is organized by submodules containing related functionality. Each submodule contains top level overview documentation, and each function contains a docstring with at least one example.

NOTE: The README on github contains information and examples complementary to these docs.

CHAPTER

ONE

THE API BY USEFULNESS

Perhaps the most useful way to learn this API is to sort by "usefulness". I measure usefulness as the number of times I've used a particular function in my own code (excluding ubelt itself).

Function name	Usefulness
ubelt.repr2	2373
ubelt.ProgIter	466
ubelt.take	343
ubelt.expandpath	338
ubelt.ensuredir	261
ubelt.paragraph	239
ubelt.map_vals	237
ubelt.cmd	234
ubelt.odict	231
ubelt.Path	222
ubelt.NiceRepr	205
ubelt.ensure_app_cache_dir	204
ubelt.iterable	201
ubelt.codeblock	200
ubelt.NoParam	196
ubelt.flatten	190
ubelt.peek	185
ubelt.ddict	172
ubelt.dzip	162
ubelt.group_items	154
ubelt.oset	132
ubelt.hash_data	128
ubelt.argflag	123
ubelt.grabdata	116
ubelt.augpath	109
ubelt.Timer	105
ubelt.dict_isect	104
ubelt.dict_diff	98
ubelt.argval	97
ubelt.allsame	96
ubelt.dict_hist	94
ubelt.delete	90
ubelt.identity	88
ubelt.color_text	85

continues on next page

Table 1 – continued from previous page

Function name	Usefulness
ubelt.compress	83
ubelt.hzcat	75
ubelt.memoize	70
ubelt.named_product	52
ubelt.dict_union	48
ubelt.map_keys	43
ubelt.invert_dict	43
ubelt.JobPool	41
ubelt.hash_file	38
ubelt.unique	38
ubelt.timestamp	38
ubelt.indent	38
ubelt.argsort	37
ubelt.IndexableWalker	36
ubelt.Cacher	36
ubelt.dict_subset	35
ubelt.iter_window	34
ubelt.memoize_property	34
ubelt.writeto	32
ubelt.sorted_vals	31
ubelt.find_exe	30
ubelt.ensure_unicode	29
ubelt.chunks	29
ubelt.find_duplicates	29
ubelt.CacheStamp	28
ubelt.memoize_method	24
ubelt.argmax	24
ubelt.import_module_from_path	24
ubelt.symlink	21
ubelt.readfrom	21
ubelt.WIN32	20
ubelt.highlight_code	20
ubelt.import_module_from_name	19
ubelt.modname_to_modpath	17
ubelt.touch	17
ubelt.get_app_cache_dir	16
ubelt.Executor	13
ubelt.sorted_keys	12
ubelt.inject_method	10
ubelt.compatible	8
ubelt.AutoDict	8
ubelt.shrinkuser	8
ubelt.find_path	7
ubelt.LINUX	7
ubelt.CaptureStdout	6
ubelt.modpath_to_modname	6
ubelt.argmin	5
ubelt.zopen	4
ubelt.varied_values	4

continues on next page

Table 1 – continued from previous page

Function name	Usefulness
ubelt.split_modpath	4
ubelt.DARWIN	4
ubelt.userhome	3
ubelt.download	3
ubelt.argunique	2
ubelt.AutoOrderedDict	1
ubelt.unique_flags	1
ubelt.split_archive	0
ubelt.platform_data_dir	0
ubelt.platform_config_dir	0
ubelt.platform_cache_dir	0
ubelt.indexable_allclose	0
ubelt.get_app_data_dir	0
ubelt.get_app_config_dir	0
ubelt.ensure_app_data_dir	0
ubelt.ensure_app_config_dir	0
ubelt.boolmask	0
ubelt.TempDir	0
ubelt.TeeStringIO	0
ubelt.POSIX	0
ubelt.OrderedSet	0
ubelt.NO_COLOR	0
ubelt.FormatterExtensions	0
ubelt.DownloadManager	0
ubelt.CaptureStream	0

```
usage stats = {
    'mean': 89.296295,
    'std': 238.03113,
    'min': 0.0,
    'max': 2373.0,
    'q_0.25': 4.75,
    'q_0.50': 30.5,
    'q_0.75': 99.5,
    'med': 30.5,
    'sum': 9644,
    'shape': (108,),
}
```

1.1 ubelt.orderedset

<ubelt.OrderedSet> *<ubelt.oset>*

1.2 ubelt.progiter

<ubelt.ProgIter>

1.3 ubelt.util_arg

<ubelt.argval> *<ubelt.argflag>*

1.4 ubelt.util_cache

<ubelt.Cacher> *<ubelt.CacheStamp>*

1.5 ubelt.util_cmd

<ubelt.cmd>

1.6 ubelt.util_colors

<ubelt.NO_COLOR> *<ubelt.highlight_code>* *<ubelt.color_text>*

1.7 ubelt.util_const

<ubelt.NoParam>

1.8 ubelt.util_dict

<ubelt.AutoDict> *<ubelt.AutoOrderedDict>* *<ubelt.dzip>* *<ubelt.ddict>* *<ubelt.dict_hist>* *<ubelt.dict_subset>* *<ubelt.dict_union>* *<ubelt.dict_isect>* *<ubelt.dict_diff>* *<ubelt.find_duplicates>* *<ubelt.group_items>* *<ubelt.invert_dict>* *<ubelt.map_keys>* *<ubelt.map_vals>* *<ubelt.sorted_keys>* *<ubelt.sorted_vals>* *<ubelt.odict>* *<ubelt.named_product>* *<ubelt.varied_values>*

1.9 ubelt.util_download

<ubelt.download> *<ubelt.grabdata>*

1.10 ubelt.util_download_manager

<ubelt.DownloadManager>

1.11 ubelt.util_format

<ubelt.repr2> *<ubelt.FormatterExtensions>*

1.12 ubelt.util_func

<ubelt.identity> *<ubelt.inject_method>* *<ubelt.compatible>*

1.13 ubelt.util_futures

<ubelt.Executor> *<ubelt.JobPool>*

1.14 ubelt.util_hash

<ubelt.hash_data> *<ubelt.hash_file>*

1.15 ubelt.util_import

<ubelt.split_modpath> *<ubelt.modname_to_modpath>* *<ubelt.modpath_to_modname>* *<ubelt.import_module_from_name>* *<ubelt.import_module_from_path>*

1.16 ubelt.util_indexable

<ubelt.IndexableWalker> *<ubelt.indexable_allclose>*

1.17 ubelt.util_io

<ubelt.readfrom> *<ubelt.writeto>* *<ubelt.touch>* *<ubelt.delete>*

1.18 ubelt.util_links

<ubelt.symlink>

1.19 ubelt.util_list

<ubelt.allsame> *<ubelt.argmax>* *<ubelt.argmin>* *<ubelt.argsort>* *<ubelt.argunique>* *<ubelt.boolmask>* *<ubelt.chunks>* *<ubelt.compress>* *<ubelt.flatten>* *<ubelt.iter_window>* *<ubelt.iterable>* *<ubelt.peek>* *<ubelt.take>* *<ubelt.unique>* *<ubelt.unique_flags>*

1.20 ubelt.util_memoize

<ubelt.memoize> *<ubelt.memoize_method>* *<ubelt.memoize_property>*

1.21 ubelt.util_mixins

<ubelt.NiceRepr>

1.22 ubelt.util_path

<ubelt.Path> *<ubelt.TempDir>* *<ubelt.augpath>* *<ubelt.shrinkuser>* *<ubelt.userhome>* *<ubelt.ensuredir>* *<ubelt.expandpath>*

1.23 ubelt.util_platform

<ubelt.WIN32> <ubelt.LINUX> <ubelt.DARWIN> <ubelt.POSIX> *<ubelt.find_exe>* *<ubelt.find_path>* *<ubelt.ensure_app_cache_dir>* *<ubelt.ensure_app_config_dir>* *<ubelt.ensure_app_data_dir>* *<ubelt.get_app_cache_dir>* *<ubelt.get_app_config_dir>* *<ubelt.get_app_data_dir>* *<ubelt.platform_cache_dir>* *<ubelt.platform_config_dir>* *<ubelt.platform_data_dir>*

1.24 ubelt.util_str

<ubelt.indent> *<ubelt.codeblock>* *<ubelt.paragraph>* *<ubelt.hzcat>* *<ubelt.ensure_unicode>*

1.25 ubelt.util_stream

<ubelt.TeeStringIO> *<ubelt.CaptureStdout>* *<ubelt.CaptureStream>*

1.26 ubelt.util_time

<ubelt.timestamp> *<ubelt.Timer>*

1.27 ubelt.util_zip

<ubelt.zopen> *<ubelt.split_archive>*

1.27.1 ubelt

1.27.1.1 ubelt package

1.27.1.1.1 Submodules

1.27.1.1.1.1 ubelt.orderedset module

This module exposes the `OrderedSet` class, which is a collection of unique items that maintains the order in which the items were added. An `OrderedSet` (or its alias `oset`) behaves very similarly to Python's builtin `set` object, the main difference being that an `OrderedSet` can efficiently lookup its items by index.

Example

```
>>> import ubelt as ub
>>> ub.oset([1, 2, 3])
OrderedSet([1, 2, 3])
>>> (ub.oset([1, 2, 3]) - {2}) | {2}
OrderedSet([1, 3, 2])
>>> [ub.oset([1, 2, 3])[i] for i in [1, 0, 2]]
[2, 1, 3]
```

As of version (0.8.5), *ubelt* contains its own internal copy of `OrderedSet` in order to reduce external dependencies. The original standalone implementation lives in https://github.com/LuminosoInsight/ordered-set.

The original documentation is as follows:

An OrderedSet is a custom MutableSet that remembers its order, so that every entry has an index that can be looked up.

Based on a recipe originally posted to ActiveState Recipes by Raymond Hettiger, and released under the MIT license.

class ubelt.orderedset.**OrderedSet**(*iterable=None*)

Bases: collections.abc.MutableSet, collections.abc.Sequence

An OrderedSet is a custom MutableSet that remembers its order, so that every entry has an index that can be looked up.

Example

```
>>> OrderedSet([1, 1, 2, 3, 2])
OrderedSet([1, 2, 3])
```

copy()

Return a shallow copy of this object.

Example

```
>>> this = OrderedSet([1, 2, 3])
>>> other = this.copy()
>>> this == other
True
>>> this is other
False
```

add(*key*)

Add *key* as an item to this OrderedSet, then return its index.

If *key* is already in the OrderedSet, return the index it already had.

Example

```
>>> oset = OrderedSet()
>>> oset.append(3)
0
>>> print(oset)
OrderedSet([3])
```

append(*key*)

Add *key* as an item to this OrderedSet, then return its index.

If *key* is already in the OrderedSet, return the index it already had.

Example

```
>>> oset = OrderedSet()
>>> oset.append(3)
0
>>> print(oset)
OrderedSet([3])
```

update(*sequence*)

Update the set with the given iterable sequence, then return the index of the last element inserted.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.update([3, 1, 5, 1, 4])
4
>>> print(oset)
OrderedSet([1, 2, 3, 5, 4])
```

index(*key*)

Get the index of a given entry, raising an IndexError if it's not present.

key can be an iterable of entries that is not a string, in which case this returns a list of indices.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.index(2)
1
```

get_loc(*key*)

Get the index of a given entry, raising an IndexError if it's not present.

key can be an iterable of entries that is not a string, in which case this returns a list of indices.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.index(2)
1
```

get_indexer(*key*)

Get the index of a given entry, raising an IndexError if it's not present.

key can be an iterable of entries that is not a string, in which case this returns a list of indices.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.index(2)
1
```

pop()

Remove and return the last element from the set.

Raises KeyError if the set is empty.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.pop()
3
```

discard(*key*)

Remove an element. Do not raise an exception if absent.

The MutableSet mixin uses this to implement the .remove() method, which *does* raise an error when asked to remove a non-existent item.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.discard(2)
>>> print(oset)
OrderedSet([1, 3])
>>> oset.discard(2)
>>> print(oset)
OrderedSet([1, 3])
```

clear()

Remove all items from this OrderedSet.

union(**sets*)

Combines all unique items. Each items order is defined by its first appearance.

Example

```
>>> oset = OrderedSet.union(OrderedSet([3, 1, 4, 1, 5]), [1, 3], [2, 0])
>>> print(oset)
OrderedSet([3, 1, 4, 5, 2, 0])
>>> oset.union([8, 9])
OrderedSet([3, 1, 4, 5, 2, 0, 8, 9])
>>> oset | {10}
OrderedSet([3, 1, 4, 5, 2, 0, 10])
```

intersection(**sets*)

Returns elements in common between all sets. Order is defined only by the first set.

Example

```
>>> oset = OrderedSet.intersection(OrderedSet([0, 1, 2, 3]), [1, 2, 3])
>>> print(oset)
OrderedSet([1, 2, 3])
>>> oset.intersection([2, 4, 5], [1, 2, 3, 4])
OrderedSet([2])
>>> oset.intersection()
OrderedSet([1, 2, 3])
```

difference(**sets*)

Returns all elements that are in this set but not the others.

Example

```
>>> OrderedSet([1, 2, 3]).difference(OrderedSet([2]))
OrderedSet([1, 3])
>>> OrderedSet([1, 2, 3]).difference(OrderedSet([2]), OrderedSet([3]))
OrderedSet([1])
>>> OrderedSet([1, 2, 3]) - OrderedSet([2])
OrderedSet([1, 3])
>>> OrderedSet([1, 2, 3]).difference()
OrderedSet([1, 2, 3])
```

issubset(*other*)

Report whether another set contains this set.

Example

```
>>> OrderedSet([1, 2, 3]).issubset({1, 2})
False
>>> OrderedSet([1, 2, 3]).issubset({1, 2, 3, 4})
True
>>> OrderedSet([1, 2, 3]).issubset({1, 4, 3, 5})
False
```

issuperset(*other*)

Report whether this set contains another set.

Example

```
>>> OrderedSet([1, 2]).issuperset([1, 2, 3])
False
>>> OrderedSet([1, 2, 3, 4]).issuperset({1, 2, 3})
True
>>> OrderedSet([1, 4, 3, 5]).issuperset({1, 2, 3})
False
```

symmetric_difference(*other*)

Return the symmetric difference of two OrderedSets as a new set. That is, the new set will contain all elements that are in exactly one of the sets.

Their order will be preserved, with elements from *self* preceding elements from *other*.

Example

```
>>> this = OrderedSet([1, 4, 3, 5, 7])
>>> other = OrderedSet([9, 7, 1, 3, 2])
>>> this.symmetric_difference(other)
OrderedSet([4, 5, 9, 2])
```

difference_update(**sets*)

Update this OrderedSet to remove items from one or more other sets.

Example

```
>>> this = OrderedSet([1, 2, 3])
>>> this.difference_update(OrderedSet([2, 4]))
>>> print(this)
OrderedSet([1, 3])
```

```
>>> this = OrderedSet([1, 2, 3, 4, 5])
>>> this.difference_update(OrderedSet([2, 4]), OrderedSet([1, 4, 6]))
>>> print(this)
OrderedSet([3, 5])
```

intersection_update(*other*)

Update this OrderedSet to keep only items in another set, preserving their order in this set.

Example

```
>>> this = OrderedSet([1, 4, 3, 5, 7])
>>> other = OrderedSet([9, 7, 1, 3, 2])
>>> this.intersection_update(other)
>>> print(this)
OrderedSet([1, 3, 7])
```

symmetric_difference_update(*other*)

Update this OrderedSet to remove items from another set, then add items from the other set that were not present in this set.

Example

```
>>> this = OrderedSet([1, 4, 3, 5, 7])
>>> other = OrderedSet([9, 7, 1, 3, 2])
>>> this.symmetric_difference_update(other)
>>> print(this)
OrderedSet([4, 5, 9, 2])
```

ubelt.orderedset.**oset**

alias of *ubelt.orderedset.OrderedSet*

1.27.1.1.1.2 ubelt.progiter module

A Progress Iterator

ProgIter lets you measure and print the progress of an iterative process. This can be done either via an iterable interface or using the manual API. Using the iterable interface is most common.

ProgIter was originally developed independently of tqdm, but the newer versions of this library have been designed to be compatible with tqdm-API. *ProgIter* is now a (mostly) drop-in alternative to `tqdm.tqdm()`. The `tqdm` library may be more appropriate in some cases. *The main advantage of :class:`ProgIter` is that it does not use any python threading*, and therefore can be safer with code that makes heavy use of multiprocessing. The reason for this is that threading before forking may cause locks to be duplicated across processes, which may lead to deadlocks.

ProgIter is simpler than tqdm, which may be desirable for some applications. However, this also means ProgIter is not as extensible as tqdm. If you want a pretty bar or need something fancy, use tqdm; if you want useful information about your iteration by default, use progiter.

The basic usage of ProgIter is simple and intuitive. Just wrap a python iterable. The following example wraps a `range` iterable and prints reported progress to stdout as the iterable is consumed.

Example

```
>>> for n in ProgIter(range(1000)):
>>>     # do some work
>>>     pass
```

Note that by default ProgIter reports information about iteration-rate, fraction-complete, estimated time remaining, time taken so far, and the current wall time.

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> def is_prime(n):
...     return n >= 2 and not any(n % i == 0 for i in range(2, n))
>>> for n in ProgIter(range(1000), verbose=1):
>>>     # do some work
>>>     is_prime(n)
1000/1000... rate=114326.51 Hz, eta=0:00:00, total=0:00:00
```

For more complex applications is may sometimes be desirable to manually use the ProgIter API. This is done as follows:

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> n = 3
>>> prog = ProgIter(desc='manual', total=n, verbose=3)
>>> prog.begin() # Manually begin progress iteration
>>> for _ in range(n):
...     prog.step(inc=1)  # specify the number of steps to increment
>>> prog.end()  # Manually end progress iteration
manual 0/3... rate=0 Hz, eta=?, total=0:00:00
manual 1/3... rate=14454.63 Hz, eta=0:00:00, total=0:00:00
```

(continues on next page)

(continued from previous page)

```
manual 2/3... rate=17485.42 Hz, eta=0:00:00, total=0:00:00
manual 3/3... rate=21689.78 Hz, eta=0:00:00, total=0:00:00
```

When working with ProgIter in either iterable or manual mode you can use the `prog.ensure_newline` method to guarantee that the next call you make to stdout will start on a new line. You can also use the `prog.set_extra` method to update a dynamci "extra" message that is shown in the formatted output. The following example demonstrates this.

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> def is_prime(n):
...     return n >= 2 and not any(n % i == 0 for i in range(2, n))
>>> _iter = range(1000)
>>> prog = ProgIter(_iter, desc='check primes', verbose=2, show_wall=True)
>>> for n in prog:
>>>     if n == 97:
>>>         print('!!! Special print at n=97 !!!')
>>>     if is_prime(n):
>>>         prog.set_extra('Biggest prime so far: {}'.format(n))
>>>         prog.ensure_newline()
check primes    0/1000... rate=0 Hz, eta=?, total=0:00:00, wall=2020-10-23 17:27 EST
check primes    1/1000... rate=95547.49 Hz, eta=0:00:00, total=0:00:00, wall=2020-10-23␣
↪17:27 EST
check primes    4/1000...Biggest prime so far: 3 rate=41062.28 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes   16/1000...Biggest prime so far: 13 rate=85340.61 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes   64/1000...Biggest prime so far: 61 rate=164739.98 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
!!! Special print at n=97 !!!
check primes  256/1000...Biggest prime so far: 251 rate=206287.91 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes  512/1000...Biggest prime so far: 509 rate=165271.92 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes  768/1000...Biggest prime so far: 761 rate=136480.12 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes 1000/1000...Biggest prime so far: 997 rate=115214.95 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
```

Todo:

- [ ] Specify callback that occurs whenever progress is written?

class ubelt.progiter.**ProgIter**(*iterable=None*, *desc=None*, *total=None*, *freq=1*, *initial=0*, *eta_window=64*, *clearline=True*, *adjust=True*, *time_thresh=2.0*, *show_times=True*, *show_wall=False*, *enabled=True*, *verbose=None*, *stream=None*, *chunksize=None*, *rel_adjust_limit=4.0*, ***kwargs*)

Bases: `ubelt.progiter._TQDMCompat`, `ubelt.progiter._BackwardsCompat`

Prints progress as an iterator progresses

ProgIter is an alternative to *tqdm*. ProgIter implements much of the tqdm-API. The main difference between *ProgIter* and *tqdm* is that ProgIter does not use threading where as *tqdm* does.

Variables

- **iterable** (*iterable*) – An iterable iterable
- **desc** (*str*) – description label to show with progress
- **total** (*int*) – Maximum length of the process. If not specified, we estimate it from the iterable, if possible.
- **freq** (*int, default=1*) – How many iterations to wait between messages.
- **adjust** (*bool, default=True*) – if True freq is adjusted based on time_thresh
- **eta_window** (*int, default=64*) – number of previous measurements to use in eta calculation
- **clearline** (*bool, default=True*) – if True messages are printed on the same line otherwise each new progress message is printed on new line.
- **adjust** – if True *freq* is adjusted based on time_thresh. This may be overwritten depending on the setting of verbose.
- **time_thresh** (*float, default=2.0*) – desired amount of time to wait between messages if adjust is True otherwise does nothing
- **show_times** (*bool, default=True*) – shows rate and eta
- **show_wall** (*bool, default=False*) – show wall time
- **initial** (*int, default=0*) – starting index offset
- **stream** (*file, default=sys.stdout*) – stream where progress information is written to
- **enabled** (*bool, default=True*) – if False nothing happens.
- **chunksize** (*int, optional*) – indicates that each iteration processes a batch of this size. Iteration rate is displayed in terms of single-items.
- **rel_adjust_limit** (*float, default=4.0*) – Maximum factor update frequency can be adjusted by in a single step.
- **verbose** (*int*) – verbosity mode, which controls clearline, adjust, and enabled. The following maps the value of *verbose* to its effect. 0: enabled=False, 1: enabled=True with clearline=True and adjust=True, 2: enabled=True with clearline=False and adjust=True, 3: enabled=True with clearline=False and adjust=False

Note: Either use ProgIter in a with statement or call prog.end() at the end of the computation if there is a possibility that the entire iterable may not be exhausted.

Note: ProgIter is an alternative to *tqdm*. The main difference between *ProgIter* and *tqdm* is that ProgIter does not use threading where as *tqdm* does. *ProgIter* is simpler than *tqdm* and thus more stable in certain circumstances.

SeeAlso: tqdm - https://pypi.python.org/pypi/tqdm

References

http://datagenetics.com/blog/february12017/index.html

Example

```
>>>
>>> def is_prime(n):
...     return n >= 2 and not any(n % i == 0 for i in range(2, n))
>>> for n in ProgIter(range(100), verbose=1, show_wall=True):
>>>     # do some work
>>>     is_prime(n)
100/100... rate=... Hz, total=..., wall=...
```

set_extra(*extra*)

specify a custom info appended to the end of the next message

---

Todo:

- [ ] extra is a bad name; come up with something better and rename

---

Example

```
>>> prog = ProgIter(range(100, 300, 100), show_times=False, verbose=3)
>>> for n in prog:
>>>     prog.set_extra('processesing num {}'.format(n))
0/2...
1/2...processesing num 100
2/2...processesing num 200
```

step(*inc=1*, *force=False*)

Manually step progress update, either directly or by an increment.

Parameters

- **inc** (*int, default=1*) – number of steps to increment
- **force** (*bool, default=False*) – if True forces progress display

Example

```
>>> n = 3
>>> prog = ProgIter(desc='manual', total=n, verbose=3)
>>> # Need to manually begin and end in this mode
>>> prog.begin()
>>> for _ in range(n):
...     prog.step()
>>> prog.end()
```

Example

```
>>> n = 3
>>> # can be used as a context manager in manual mode
>>> with ProgIter(desc='manual', total=n, verbose=3) as prog:
...     for _ in range(n):
...         prog.step()
```

start()

Alias of *ubelt.progiter.ProgIter.begin()*

begin()

Initializes information used to measure progress

This only needs to be used if this ProgIter is not wrapping an iterable. Does nothing if the this ProgIter is disabled.

Returns a chainable self-reference

Return type *ProgIter*

end()

Signals that iteration has ended and displays the final message.

This only needs to be used if this ProgIter is not wrapping an iterable. Does nothing if the this ProgIter object is disabled or has already finished.

format_message()

builds a formatted progress message with the current values. This contains the special characters needed to clear lines.

Example

```
>>> self = ProgIter(clearline=False, show_times=False)
>>> print(repr(self.format_message()))
'    0/?... \n'
>>> self.begin()
>>> self.step()
>>> print(repr(self.format_message()))
' 1/?... \n'
```

Example

```
>>> self = ProgIter(chunksize=10, total=100, clearline=False,
>>>                 show_times=False, microseconds=True)
>>> # hack, microseconds=True for coverage, needs real test
>>> print(repr(self.format_message()))
' 0.00% of 10x100... \n'
>>> self.begin()
>>> self.update()  # tqdm alternative to step
>>> print(repr(self.format_message()))
' 1.00% of 10x100... \n'
```

ensure_newline()

use before any custom printing when using the progress iter to ensure your print statement starts on a new line instead of at the end of a progress line

Example

```
>>> # Unsafe version may write your message on the wrong line
>>> prog = ProgIter(range(3), show_times=False, freq=2, adjust=False)
>>> for n in prog:
...     print('unsafe message')
 0/3... unsafe message
unsafe message
 2/3... unsafe message
 3/3...
>>> # apparently the safe version does this too.
>>> print('---')
---
>>> prog = ProgIter(range(3), show_times=False, freq=2, adjust=False)
>>> for n in prog:
...     prog.ensure_newline()
...     print('safe message')
 0/3...
safe message
safe message
 2/3...
safe message
 3/3...
```

display_message()

Writes current progress to the output stream

1.27.1.1.1.3 ubelt.util_arg module

Simple ways to interact with the commandline without defining a full blown CLI. These are usually used for developer hacks. Any real interface should probably be defined using `argparse` or `click`. Be sure to ignore unknown arguments if you use them in conjunction with these functions.

The *argflag()* function checks if a boolean `--flag` style CLI argument exists on the command line.

The *argval()* function returns the value of a `--key=value` style CLI argument.

`ubelt.util_arg.`**argval**(*key*, *default=NoParam*, *argv=None*)

Get the value of a keyword argument specified on the command line.

Values can be specified as `<key> <value>` or `<key>=<value>`

Parameters

- **key** (*str | Tuple[str, ...]*) – string or tuple of strings. Each key should be prefixed with two hyphens (i.e. `--`)
- **default** (*T | util_const._NoParamType, default=NoParam*) – a value to return if not specified.
- **argv** (*Optional[List[str]], default=None*) – uses `sys.argv` if unspecified

Returns value - the value specified after the key. It they key is specified multiple times, then the first value is returned.

Return type str | T

Todo:

- [ ] Can we handle the case where the value is a list of long paths?
- [ ] Should we default the first or last specified instance of the flag.

Example

```
>>> import ubelt as ub
>>> argv = ['--ans', '42', '--quest=the grail', '--ans=6', '--bad']
>>> assert ub.argval('--spam', argv=argv) == ub.NoParam
>>> assert ub.argval('--quest', argv=argv) == 'the grail'
>>> assert ub.argval('--ans', argv=argv) == '42'
>>> assert ub.argval('--bad', argv=argv) == ub.NoParam
>>> assert ub.argval(('--bad', '--bar'), argv=argv) == ub.NoParam
```

Example

```
>>> # Test fix for GH Issue #41
>>> import ubelt as ub
>>> argv = ['--path=/path/with/k=3']
>>> ub.argval('--path', argv=argv) == '/path/with/k=3'
```

`ubelt.util_arg.`**`argflag`**(*key*, *argv=None*)

Determines if a key is specified on the command line.

This is a functional alternative to `key in sys.argv`.

Parameters

- **key** (*str | Tuple[str, ...]*) – string or tuple of strings. Each key should be prefixed with two hyphens (i.e. `--`).
- **argv** (*List[str], default=None*) – overrides `sys.argv` if specified

Returns flag - True if the key (or any of the keys) was specified

Return type bool

Example

```
>>> import ubelt as ub
>>> argv = ['--spam', '--eggs', 'foo']
>>> assert ub.argflag('--eggs', argv=argv) is True
>>> assert ub.argflag('--ans', argv=argv) is False
>>> assert ub.argflag('foo', argv=argv) is True
>>> assert ub.argflag(('bar', '--spam'), argv=argv) is True
```

1.27.1.1.1.4 ubelt.util_cache module

This module exposes *Cacher* and *CacheStamp* classes, which provide a simple API for on-disk caching.

The *Cacher* class is the simplest and most direct method of caching. In fact, it only requires four lines of boilerplate, which is the smallest general and robust way that I (Jon Crall) have achieved, and I don't think its possible to do better. These four lines implement the following necessary and sufficient steps for general robust on-disk caching.

1. Defining the cache dependencies
2. Checking if the cache missed
3. Loading the cache on a hit
4. Executing the process and saving the result on a miss.

The following example illustrates these four points.

Example

```
>>> import ubelt as ub
>>> # Define a cache name and dependencies (which is fed to `ub.hash_data`)
>>> cacher = ub.Cacher('name', depends='set-of-deps')  # boilerplate:1
>>> # Calling tryload will return your data on a hit and None on a miss
>>> data = cacher.tryload(on_error='clear')            # boilerplate:2
>>> # Check if you need to recompute your data
>>> if data is None:                                   # boilerplate:3
>>>     # Your code to recompute data goes here (this is not boilerplate).
>>>     data = 'mydata'
>>>     # Cache the computation result (via pickle)
>>>     cacher.save(data)                               # boilerplate:4
```

Surprisingly this uses just as many boilerplate lines as a decorator style cacher, but it is much more extensible. It is possible to use *Cacher* in more sophisticated ways (e.g. metadata), but the simple in-line use is often easier and cleaner. The following example illustrates this:

Example

```
>>> import ubelt as ub
```

```
>>> @ub.Cacher('name', depends={'dep1': 1, 'dep2': 2})  # boilerplate:1
>>> def func():                                         # boilerplate:2
>>>     data = 'mydata'
>>>     return data                                     # boilerplate:3
>>> data = func()                                       # boilerplate:4
```

```
>>> cacher = ub.Cacher('name', depends=['dependencies'])  # boilerplate:1
>>> data = cacher.tryload(on_error='clear')               # boilerplate:2
>>> if data is None:                                      # boilerplate:3
>>>     data = 'mydata'
>>>     cacher.save(data)                                 # boilerplate:4
```

While the above two are equivalent, the second version provides a simpler traceback, explicit procedures, and makes it easier to use breakpoint debugging (because there is no closure scope).

While *Cacher* is used to store direct results of in-line code in a pickle format, the *CacheStamp* object is used to cache processes that produces an on-disk side effects other than the main return value. For instance, consider the following example:

Example

```
>>> import ubelt as ub
>>> def compute_many_files(dpath):
...     for i in range(10):
...         fpath = '{}/file{}.txt'.format(dpath, i)
...         with open(fpath, 'w') as file:
...             file.write('foo' + str(i))
>>> dpath = ub.Path.appdir('ubelt/demo/cache').delete().ensuredir()
>>> # You must specify a directory, unlike in Cacher where it is optional
>>> self = ub.CacheStamp('name', dpath=dpath, depends={'a': 1, 'b': 2})
>>> if self.expired():
>>>     compute_many_files(dpath)
>>>     # Instead of caching the whole processes, we just write a file
>>>     # that signals the process has been done.
>>>     self.renew()
>>> assert not self.expired()
```

The CacheStamp is lightweight in that it simply marks that a process has been completed, but the job of saving / loading the actual data is left to the developer. The `expired` method checks if the stamp exists, and `renew` writes the stamp to disk.

In ubelt version 1.1.0, several additional features were added to CacheStamp. In addition to specifying parameters via `depends`, it is also possible for CacheStamp to determine if an associated file has been modified. To do this, the paths of the files must be known a-priori and passed to CacheStamp via the `product` argument. This will allow the CacheStamp to detect if the files have been modified since the `renew` method was called. It does this by remembering the size, modified time, and checksum of each file. If the hash of the expected hash of the product is known in advance, it is also possible to specify the expected `hash_prefix` of each product. In this case, `renew` will raise an Exception if this specified hash prefix does not match the files on disk. Lastly, it is possible to specify an expiration time via

`expires`, after which the CacheStamp will always be marked as invalid. This is now the mechanism via which the cache in *`ubelt.util_download.grabdata()`* works.

Example

```
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('ubelt/demo/cache').delete().ensuredir()
>>> params = {'a': 1, 'b': 2}
>>> expected_fpaths = [dpath / 'file{}.txt'.format(i) for i in range(2)]
>>> hash_prefix = ['a7a8a91659601590e17191301dc1',
...                '55ae75d991c770d8f3ef07cbfde1']
>>> self = ub.CacheStamp('name', dpath=dpath, depends=params,
>>>                      hash_prefix=hash_prefix, hasher='sha256',
>>>                      product=expected_fpaths, expires='2101-01-01T000000Z')
>>> if self.expired():
>>>     for fpath in expected_fpaths:
...         fpath.write_text(fpath.name)
>>>     self.renew()
>>> # modifying or removing the file will cause the stamp to expire
>>> expected_fpaths[0].write_text('corrupted')
>>> assert self.expired()
```

class ubelt.util_cache.**Cacher**(*fname*, *depends=None*, *dpath=None*, *appname='ubelt'*, *ext='.pkl'*, *meta=None*, *verbose=None*, *enabled=True*, *log=None*, *hasher='sha1'*, *protocol=- 1*, *cfgstr=None*, *backend='auto'*)

Bases: `object`

Saves data to disk and reloads it based on specified dependencies.

Cacher uses pickle to save/load data to/from disk. Dependencies of the cached process can be specified, which ensures the cached data is recomputed if the dependencies change. If the location of the cache is not specified, it will default to the system user's cache directory.

Parameters

- **fname** (*str*) – A file name. This is the prefix that will be used by the cache. It will always be used as-is.
- **depends** (*str | List[str] | None*) – Indicate dependencies of this cache. If the dependencies change, then the cache is recomputed. New in version 0.8.9, replaces `cfgstr`.
- **dpath** (*str | PathLike | None*) – Specifies where to save the cache. If unspecified, Cacher defaults to an application cache dir as given by appname. See `ub.get_app_cache_dir()` for more details.
- **appname** (*str, default='ubelt'*) – Application name Specifies a folder in the application cache directory where to cache the data if `dpath` is not specified.
- **ext** (*str, default='.pkl'*) – File extension for the cache format. Can be `'.pkl'` or `'.json'`.
- **meta** (*object | None*) – Metadata that is also saved with the `cfgstr`. This can be useful to indicate how the `cfgstr` was constructed. Note: this is a candidate for deprecation.
- **verbose** (*int, default=1*) – Level of verbosity. Can be 1, 2 or 3.
- **enabled** (*bool, default=True*) – If set to False, then the load and save methods will do nothing.
- **log** (*Callable[[str], Any]*) – Overloads the print function. Useful for sending output to loggers (e.g. logging.info, tqdm.tqdm.write, ...)

- **hasher** (*str, default='sha1'*) – Type of hashing algorithm to use if `cfgstr` needs to be condensed to less than 49 characters.
- **protocol** (*int, default=-1*) – Protocol version used by pickle. Defaults to the -1 which is the latest protocol.
- **backend** (*str*) – Set to either `'pickle'` or `'json'` to force backend. Defaults to auto which chooses one based on the extension.
- **cfgstr** (*str | None*) – Deprecated in favor of `depends`.

Example

```
>>> import ubelt as ub
>>> depends = 'repr-of-params-that-uniquely-determine-the-process'
>>> # Create a cacher and try loading the data
>>> cacher = ub.Cacher('demo_process', depends, verbose=4)
>>> cacher.clear()
>>> print(f'cacher.fpath={cacher.fpath}')
>>> data = cacher.tryload()
>>> if data is None:
>>>     # Put expensive functions in if block when cacher misses
>>>     myvar1 = 'result of expensive process'
>>>     myvar2 = 'another result'
>>>     # Tell the cacher to write at the end of the if block
>>>     # It is idomatic to put results in an object named data
>>>     data = myvar1, myvar2
>>>     cacher.save(data)
>>> # Last part of the Cacher pattern is to unpack the data object
>>> myvar1, myvar2 = data
>>> #
>>> # If we know the data exists, we can also simply call load
>>> data = cacher.tryload()
```

Example

```
>>> # The previous example can be shorted if only a single value
>>> from ubelt.util_cache import Cacher
>>> depends = 'repr-of-params-that-uniquely-determine-the-process'
>>> # Create a cacher and try loading the data
>>> cacher = Cacher('demo_process', depends)
>>> myvar = cacher.tryload()
>>> if myvar is None:
>>>     myvar = ('result of expensive process', 'another result')
>>>     cacher.save(myvar)
>>> assert cacher.exists(), 'should now exist'
```

VERBOSE = 1

FORCE_DISABLE = False

property fpath

get_fpath(*cfgstr=None*)

Reports the filepath that the cacher will use.

It will attempt to use '{fname}_{cfgstr}{ext}' unless that is too long. Then cfgstr will be hashed.

Parameters **cfgstr** (*str | None*) – overrides the instance-level cfgstr

Returns str | PathLike

Example

```
>>> # xdoctest: +REQUIRES(module:pytest)
>>> from ubelt.util_cache import Cacher
>>> import pytest
>>> with pytest.warns(UserWarning):
>>>     cacher = Cacher('test_cacher1')
>>>     cacher.get_fpath()
>>> self = Cacher('test_cacher2', depends='cfg1')
>>> self.get_fpath()
>>> self = Cacher('test_cacher3', depends='cfg1' * 32)
>>> self.get_fpath()
```

exists(*cfgstr=None*)

Check to see if the cache exists

Parameters **cfgstr** (*str | None*) – overrides the instance-level cfgstr

Returns bool

existing_versions()

Returns data with different cfgstr values that were previously computed with this cacher.

Yields *str* – paths to cached files corresponding to this cacher

Example

```
>>> from ubelt.util_cache import Cacher
>>> # Ensure that some data exists
>>> known_fpaths = set()
>>> import ubelt as ub
>>> dpath = ub.ensure_app_cache_dir('ubelt',
>>>                                 'test-existing-versions')
>>> ub.delete(dpath)  # start fresh
>>> cacher = Cacher('versioned_data_v2', depends='1', dpath=dpath)
>>> cacher.ensure(lambda: 'data1')
>>> known_fpaths.add(cacher.get_fpath())
>>> cacher = Cacher('versioned_data_v2', depends='2', dpath=dpath)
>>> cacher.ensure(lambda: 'data2')
>>> known_fpaths.add(cacher.get_fpath())
>>> # List previously computed configs for this type
>>> from os.path import basename
>>> cacher = Cacher('versioned_data_v2', depends='2', dpath=dpath)
>>> exist_fpaths = set(cacher.existing_versions())
>>> exist_fnames = list(map(basename, exist_fpaths))
```

(continues on next page)

(continued from previous page)

```
>>> print('exist_fnames = {!r}'.format(exist_fnames))
>>> assert exist_fpaths.issubset(known_fpaths)
```

clear(*cfgstr=None*)

Removes the saved cache and metadata from disk

Parameters **cfgstr** (*str | None*) – overrides the instance-level cfgstr

tryload(*cfgstr=None*, *on_error='raise'*)

Like load, but returns None if the load fails due to a cache miss.

Parameters

- **cfgstr** (*str | None*) – overrides the instance-level cfgstr
- **on_error** (*str, default='raise'*) – How to handle non-io errors errors. Either 'raise', which re-raises the exception, or 'clear' which deletes the cache and returns None.

Returns the cached data if it exists, otherwise returns None

Return type None | object

load(*cfgstr=None*)

Load the data cached and raise an error if something goes wrong.

Parameters **cfgstr** (*str | None*) – overrides the instance-level cfgstr

Returns the cached data

Return type object

Raises **IOError - if the data is unable to be loaded. This could be due to** – a cache miss or because the cache is disabled.

Example

```
>>> from ubelt.util_cache import *  # NOQA
>>> # Setting the cacher as enabled=False turns it off
>>> cacher = Cacher('test_disabled_load', '', enabled=True)
>>> cacher.save('data')
>>> assert cacher.load() == 'data'
>>> cacher.enabled = False
>>> assert cacher.tryload() is None
```

save(*data*, *cfgstr=None*)

Writes data to path specified by `self.fpath`.

Metadata containing information about the cache will also be appended to an adjacent file with the *.meta* suffix.

Parameters

- **data** (*object*) – arbitrary pickleable object to be cached
- **cfgstr** (*str | None*) – overrides the instance-level cfgstr

Example

```
>>> from ubelt.util_cache import *  # NOQA
>>> # Normal functioning
>>> depends = 'long-cfg' * 32
>>> cacher = Cacher('test_enabled_save', depends=depends)
>>> cacher.save('data')
>>> assert exists(cacher.get_fpath()), 'should be enabled'
>>> assert exists(cacher.get_fpath() + '.meta'), 'missing metadata'
>>> # Setting the cacher as enabled=False turns it off
>>> cacher2 = Cacher('test_disabled_save', 'params', enabled=False)
>>> cacher2.save('data')
>>> assert not exists(cacher2.get_fpath()), 'should be disabled'
```

ensure(*func*, **args*, ***kwargs*)

Wraps around a function. A cfgstr must be stored in the base cacher.

Parameters

- **func** (*Callable*) – function that will compute data on cache miss
- ***args** – passed to func
- ****kwargs** – passed to func

Example

```
>>> from ubelt.util_cache import *  # NOQA
>>> def func():
>>>     return 'expensive result'
>>> fname = 'test_cacher_ensure'
>>> depends = 'func params'
>>> cacher = Cacher(fname, depends=depends)
>>> cacher.clear()
>>> data1 = cacher.ensure(func)
>>> data2 = cacher.ensure(func)
>>> assert data1 == 'expensive result'
>>> assert data1 == data2
>>> cacher.clear()
```

class ubelt.util_cache.**CacheStamp**(*fname*, *dpath*, *cfgstr=None*, *product=None*, *hasher='sha1'*, *verbose=None*, *enabled=True*, *depends=None*, *meta=None*, *hash_prefix=None*, *expires=None*, *ext='.pkl'*)

Bases: `object`

Quickly determine if a file-producing computation has been done.

Check if the computation needs to be redone by calling `expired`. If the stamp is not expired, the user can expect that the results exist and could be loaded. If the stamp is expired, the computation should be redone. After the result is updated, the calls `renew`, which writes a "stamp" file to disk that marks that the procedure has been done.

There are several ways to control how a stamp expires. At a bare minimum, removing the stamp file will force expiration. However, in this circumstance CacheStamp only knows that something has been done, but it doesn't have any information about what was done, so in general this is not sufficient.

To achieve more robust expiration behavior, the user should specify the `product` argument, which is a list of file paths that are expected to exist whenever the stamp is renewed. When this is specified the CacheStamp will expire if any of these products are deleted, their size changes, their modified timestamp changes, or their hash (i.e. checksum) changes. Note that by setting `hasher=None`, running and verifying checksums can be disabled.

If the user knows what the hash of the file should be this can be specified to prevent renewal of the stamp unless these match the files on disk. This can be useful for security purposes.

The stamp can also be set to expire at a specified time or after a specified duration using the `expires` argument.

Parameters

- **fname** (*str*) – Name of the stamp file
- **dpath** (*str | PathLike | None*) – Where to store the cached stamp file
- **product** (*str | PathLike | Sequence[str | PathLike] | None*) – Path or paths that we expect the computation to produce. If specified the hash of the paths are stored.
- **hasher** (*str, default='sha1'*) – The type of hasher used to compute the file hash of product. If None, then we assume the file has not been corrupted or changed if the mtime and size are the same. Defaults to sha1.
- **verbose** (*bool, default=None*) – Passed to internal `ubelt.Cacher` object
- **enabled** (*bool, default=True*) – if False, expired always returns True
- **depends** (*str | List[str] | None*) – Indicate dependencies of this cache. If the dependencies change, then the cache is recomputed. New to CacheStamp in version 0.9.2, replaces `cfgstr`.
- **meta** (*object | None*) – Metadata that is also saved with the `cfgstr`. This can be useful to indicate how the `cfgstr` was constructed. New to CacheStamp in version 0.9.2. Note: this is a candidate for deprecation.
- **expires** (*str | int | datetime.datetime | datetime.timedelta | None*) – If specified, sets an expiration date for the certificate. This can be an absolute datetime or a timedelta offset. If specified as an int, this is interpreted as a time delta in seconds. If specified as a str, this is interpreted as an absolute timestamp. Time delta offsets are coerced to absolute times at "renew" time.
- **hash_prefix** (*None | str | List[str]*) – If specified, we verify that these match the hash(s) of the product(s) in the stamp certificate.
- **ext** (*str, default='.pkl'*) – File extension for the cache format. Can be `'.pkl'` or `'.json'`.
- **cfgstr** (*str | None*) – DEPRECATED in favor or depends.

Example

```
>>> import ubelt as ub
>>> # Stamp the computation of expensive-to-compute.txt
>>> dpath = ub.Path.appdir('ubelt/tests/cache-stamp')
>>> dpath.delete().ensuredir()
>>> product = dpath / 'expensive-to-compute.txt'
>>> self = ub.CacheStamp('somedata', depends='someconfig', dpath=dpath,
>>>                      product=product, hasher='sha256')
>>> self.clear()
>>> print(f'self.fpath={self.fpath}')
>>> if self.expired():
>>>     product.write_text('very expensive')
```

(continues on next page)

(continued from previous page)

```
>>>     self.renew()
>>> assert not self.expired()
>>> # corrupting the output will cause the stamp to expire
>>> product.write_text('very corrupted')
>>> assert self.expired()
```

property fpath

clear()

Delete the stamp (the products are untouched)

expired(*cfgstr=None*, *product=None*)

Check to see if a previously existing stamp is still valid, if the expected result of that computation still exists, and if all other expiration criteria are met.

Parameters

- **cfgstr** (*str | None*) – overrides the instance-level cfgstr. DEPRECATED do not use.
- **product** (*str | PathLike | Sequence[str | PathLike] | None*) – override the default product if specified. DEPRECATED do not use.

Returns True(-thy) if the stamp is invalid, expired, or does not exist. When the stamp is expired, the reason for expiration is returned as a string. If the stamp is still valid, False is returned.

Return type bool | str

Example

```
>>> import ubelt as ub
>>> import time
>>> import os
>>> # Stamp the computation of expensive-to-compute.txt
>>> dpath = ub.Path.appdir('ubelt/tests/cache-stamp-expired')
>>> dpath.delete().ensuredir()
>>> products = [
>>>     dpath / 'product1.txt',
>>>     dpath / 'product2.txt',
>>> ]
>>> self = ub.CacheStamp('myname', depends='myconfig', dpath=dpath,
>>>                      product=products, hasher='sha256',
>>>                      expires=0)
>>> if self.expired():
>>>     for fpath in products:
>>>         fpath.write_text(fpath.name)
>>>     self.renew()
>>> fpath = products[0]
>>> # Because we set the expiration delta to 0, we should already be expired
>>> assert self.expired() == 'expired_cert'
>>> # Disable the expiration date, renew and we should be ok
>>> self.expires = None
>>> self.renew()
>>> assert not self.expired()
>>> # Modify the mtime to cause expiration
```

(continues on next page)

(continued from previous page)

```
>>> orig_atime = fpath.stat().st_atime
>>> orig_mtime = fpath.stat().st_mtime
>>> os.utime(fpath, (orig_atime, orig_mtime + 200))
>>> assert self.expired() == 'mtime_diff'
>>> self.renew()
>>> assert not self.expired()
>>> # rewriting the file will cause the size constraint to fail
>>> # even if we hack the mtime to be the same
>>> orig_atime = fpath.stat().st_atime
>>> orig_mtime = fpath.stat().st_mtime
>>> fpath.write_text('corrupted')
>>> os.utime(fpath, (orig_atime, orig_mtime))
>>> assert self.expired() == 'size_diff'
>>> self.renew()
>>> assert not self.expired()
>>> # Force a situation where the hash is the only thing
>>> # that saves us, write a different file with the same
>>> # size and mtime.
>>> orig_atime = fpath.stat().st_atime
>>> orig_mtime = fpath.stat().st_mtime
>>> fpath.write_text('corrApted')
>>> os.utime(fpath, (orig_atime, orig_mtime))
>>> assert self.expired() == 'hash_diff'
>>> # Test what a wrong hash prefix causes expiration
>>> certificate = self.renew()
>>> self.hash_prefix = certificate['hash']
>>> self.expired()
>>> self.hash_prefix = ['bad', 'hashes']
>>> self.expired()
>>> # A bad hash will not allow us to renew
>>> import pytest
>>> with pytest.raises(RuntimeError):
...     self.renew()
```

renew(*cfgstr=None*, *product=None*)

Recertify that the product has been recomputed by writing a new certificate to disk.

Returns certificate information

Return type dict

1.27.1.1.1.5 ubelt.util_cmd module

This module exposes the *ubelt.cmd()* command, which provides a simple means for interacting with the command-line. This uses `subprocess.Popen` under the hood, but improves upon existing `subprocess` functionality by:

(1) Adding the option to "tee" the output, i.e. simultaniously capture and write to stdout and stderr.

(2) Always specify the command as a string. The `subprocess` module expects the command as either a `List[str]` if `shell=False` and `str` if `shell=True`. If necessary, *ubelt.util_cmd.cmd()* will automatically convert from one format to the other, so passing in either case will work.

(3) Specificy if the process blocks or not by setting `detach`. Note: when `detach is True` it is not possible to tee the output.

Example

```
>>> import ubelt as ub
>>> # Running with verbose=1 will write to stdout in real time
>>> info = ub.cmd('echo "write your command naturally"', verbose=1)
write your command naturally
>>> # Unless `detach=True`, `cmd` always returns an info dict.
>>> print('info = ' + ub.repr2(info))
info = {
    'command': 'echo "write your command naturally"',
    'cwd': None,
    'err': '',
    'out': 'write your command naturally\n',
    'proc': <...Popen...>,
    'ret': 0,
}
```

`ubelt.util_cmd.cmd`(*command*, *shell=False*, *detach=False*, *verbose=0*, *tee=None*, *cwd=None*, *env=None*, *tee_backend='auto'*, *check=False*, *system=False*, *timeout=None*)

Executes a command in a subprocess.

The advantage of this wrapper around subprocess is that (1) you control if the subprocess prints to stdout, (2) the text written to stdout and stderr is returned for parsing, (3) cross platform behavior that lets you specify the command as a string or tuple regardless of whether or not shell=True. (4) ability to detach, return the process object and allow the process to run in the background (eventually we may return a Future object instead).

Parameters

- **command** (*str | List[str]*) – bash-like command string or tuple of executable and args
- **shell** (*bool, default=False*) – if True, process is run in shell.
- **detach** (*bool, default=False*) – if True, process is detached and run in background.
- **verbose** (*int, default=0*) – verbosity mode. Can be 0, 1, 2, or 3.
- **tee** (*bool | None*) – if True, simultaneously writes to stdout while capturing output from the command. If not specified, defaults to True if verbose > 0. If detach is True, then this argument is ignored.
- **cwd** (*str | PathLike | None*) – Path to run command. Defaults to current working directory if unspecified.
- **env** (*Dict[str, str] | None*) – environment passed to Popen
- **tee_backend** (*str, default='auto'*) – backend for tee output. Valid choices are: "auto", "select" (POSIX only), and "thread".
- **check** (*bool, default=False*) – if True, check that the return code was zero before returning, otherwise raise a CalledProcessError. Does nothing if detach is True.
- **system** (*bool, default=False*) – if True, most other considerations are dropped, and `os.system()` is used to execute the command in a platform dependant way. Other arguments such as env, tee, timeout, and shell are all ignored. (new in version 1.1.0)
- **timeout** (*float*) – If the process does not complete in *timeout* seconds, raises a `subprocess.TimeoutExpired`. (new in version 1.1.0)

Returns info - information about command status. if detach is False `info` contains captured standard out, standard error, and the return code if detach is False `info` contains a reference to the process.

Return type dict

Note: Inputs can either be text or tuple based. On UNIX we ensure conversion to text if shell=True, and to tuple if shell=False. On windows, the input is always text based. See [SO_33560364] for a potential cross-platform shlex solution for windows.

When using the tee output, the stdout and stderr may be shuffled from what they would be on the command line.

CommandLine

```
xdoctest -m ubelt.util_cmd cmd:6
python -c "import ubelt as ub; ub.cmd('ping localhost -c 2', verbose=2)"
pytest "$(python -c 'import ubelt; print(ubelt.util_cmd.__file__)')" -sv --xdoctest-
↪verbose 2
```

References

Example

```
>>> import ubelt as ub
>>> info = ub.cmd(('echo', 'simple cmdline interface'), verbose=1)
simple cmdline interface
>>> assert info['ret'] == 0
>>> assert info['out'].strip() == 'simple cmdline interface'
>>> assert info['err'].strip() == ''
```

Example

```
>>> import ubelt as ub
>>> info = ub.cmd('echo str noshell', verbose=0)
>>> assert info['out'].strip() == 'str noshell'
```

Example

```
>>> # windows echo will output extra single quotes
>>> import ubelt as ub
>>> info = ub.cmd(('echo', 'tuple noshell'), verbose=0)
>>> assert info['out'].strip().strip("'") == 'tuple noshell'
```

Example

```
>>> # Note this command is formatted to work on win32 and unix
>>> import ubelt as ub
>>> info = ub.cmd('echo str&&echo shell', verbose=0, shell=True)
>>> assert info['out'].strip() == 'str' + chr(10) + 'shell'
```

Example

```
>>> import ubelt as ub
>>> info = ub.cmd(('echo', 'tuple shell'), verbose=0, shell=True)
>>> assert info['out'].strip().strip("'") == 'tuple shell'
```

Example

```
>>> import pytest
>>> import ubelt as ub
>>> info = ub.cmd('echo hi', check=True)
>>> import subprocess
>>> with pytest.raises(subprocess.CalledProcessError):
>>>     ub.cmd('exit 1', check=True, shell=True)
```

Example

```
>>> import ubelt as ub
>>> from os.path import join, exists
>>> fpath1 = join(ub.get_app_cache_dir('ubelt'), 'cmdout1.txt')
>>> fpath2 = join(ub.get_app_cache_dir('ubelt'), 'cmdout2.txt')
>>> ub.delete(fpath1)
>>> ub.delete(fpath2)
>>> # Start up two processes that run simultaneously in the background
>>> info1 = ub.cmd(('touch', fpath1), detach=True)
>>> info2 = ub.cmd('echo writing2 > ' + fpath2, shell=True, detach=True)
>>> # Detached processes are running in the background
>>> # We can run other code while we wait for them.
>>> while not exists(fpath1):
...     pass
>>> while not exists(fpath2):
...     pass
>>> # communicate with the process before you finish
>>> # (otherwise you may leak a text wrapper)
>>> info1['proc'].communicate()
>>> info2['proc'].communicate()
>>> # Check that the process actually did finish
>>> assert (info1['proc'].wait()) == 0
>>> assert (info2['proc'].wait()) == 0
>>> # Check that the process did what we expect
>>> assert ub.readfrom(fpath1) == ''
>>> assert ub.readfrom(fpath2).strip() == 'writing2'
```

Example

```
>>> # Can also use ub.cmd to call os.system
>>> import pytest
>>> import ubelt as ub
>>> import subprocess
>>> info = ub.cmd('echo hi', check=True, system=True)
>>> with pytest.raises(subprocess.CalledProcessError):
>>>     ub.cmd('exit 1', check=True, shell=True)
```

1.27.1.1.1.6 ubelt.util_colors module

This module defines simple functions to color your text and highlight your code using ANSI escape sequences. This works using the Pygments library, which is an optional requirement. Therefore, these functions only work properly if Pygments is installed, otherwise these functions will return the unmodified text and a warning will be printed.

The *highlight_code()* function uses pygments to highlight syntax of a programming language.

The *color_text()* function colors text with a solid color.

Note the functions in this module require the optional `pygments` library to work correctly. These functions will warn if `pygments` is not installed.

This module contains a global variable `NO_COLOR`, which if set to True will force all ANSI text coloring functions to become no-ops. This defaults to the value of the `bool(os.environ.get('NO_COLOR'))` flag, which is compliant with [NoColor].

References

Note: In the future we may rename this module to `util_ansi`.

Requirements: pip install pygments

`ubelt.util_colors.`**`highlight_code`**(*text*, *lexer_name='python'*, ***kwargs*)

Highlights a block of text using ANSI tags based on language syntax.

Parameters

- **text** (*str*) – plain text to highlight
- **lexer_name** (*str*) – name of language. eg: python, docker, c++
- ****kwargs** – passed to pygments.lexers.get_lexer_by_name

Returns text - highlighted text If pygments is not installed, the plain text is returned.

Return type str

Example

```
>>> import ubelt as ub
>>> text = 'import ubelt as ub; print(ub)'
>>> new_text = ub.highlight_code(text)
>>> print(new_text)
```

`ubelt.util_colors.color_text(text, color)`

Colorizes text a single color using ansi tags.

Parameters

- **text** (*str*) – text to colorize
- **color** (*str*) – color code. different systems may have different colors. commonly available colors are: 'red', 'brown', 'yellow', 'green', 'blue', 'black', and 'white'.

Returns text - colorized text. If pygments is not installed plain text is returned.

Return type str

Example

```
>>> text = 'raw text'
>>> import pytest
>>> import ubelt as ub
>>> if ub.modname_to_modpath('pygments'):
>>>     # Colors text only if pygments is installed
>>>     ansi_text = ub.ensure_unicode(color_text(text, 'red'))
>>>     prefix = ub.ensure_unicode('\x1b[31')
>>>     print('prefix = {!r}'.format(prefix))
>>>     print('ansi_text = {!r}'.format(ansi_text))
>>>     assert ansi_text.startswith(prefix)
>>>     assert color_text(text, None) == 'raw text'
>>> else:
>>>     # Otherwise text passes through unchanged
>>>     assert color_text(text, 'red') == 'raw text'
>>>     assert color_text(text, None) == 'raw text'
```

1.27.1.1.1.7 ubelt.util_const module

This module defines `ub.NoParam`. This is a robust sentinel value that can act like `None` when None might be a valid value. The value of `NoParam` is robust to reloading, pickling, and copying (i.e. `var is ub.NoParam` will return `True` after these operations).

Use cases that demonstrate the value of `NoParam` can be found in *`ubelt.util_dict`*, where it simplifies the implementation of methods that behave like `dict.get()`.

Example

```
>>> import ubelt as ub
>>> def func(a=ub.NoParam):
>>>     if a is ub.NoParam:
>>>         print('no param specified')
>>>     else:
>>>         print('a = {}'.format(a))
>>> func()
no param specified
>>> func(a=None)
a = None
>>> func(a=1)
a = 1
>>> # note: typically it is bad practice to use NoParam as an actual
>>> # (non-default) parameter. It goes against the sprit of the idea.
>>> func(a=ub.NoParam)
no param specified
```

1.27.1.1.1.8 ubelt.util_dict module

Functions for working with dictionaries.

The *dict_hist()* function counts the number of discrete occurrences of hashable items. Similarly *find_duplicates()* looks for indices of items that occur more than *k=1* times.

The *map_keys()* and *map_vals()* functions are useful for transforming the keys and values of a dictionary with less syntax than a dict comprehension.

The *dict_union()*, *dict_isect()*, and *dict_diff()* functions are similar to the set equivalents.

The *dzip()* function zips two iterables and packs them into a dictionary where the first iterable is used to generate keys and the second generates values.

The *group_items()* function takes two lists and returns a dict mapping values in the second list to all items in corresponding locations in the first list.

The *invert_dict()* function swaps keys and values. See the function docs for details on dealing with unique and non-unique values.

The *ddict()* and *odict()* functions are alias for the commonly used `collections.defaultdict()` and `collections.OrderedDict()` classes.

class ubelt.util_dict.**AutoDict**

Bases: `dict`

An infinitely nested default dict of dicts.

Implementation of Perl's autovivification feature.

SeeAlso: *AutoOrderedDict* - the ordered version

References

Example

```
>>> import ubelt as ub
>>> auto = ub.AutoDict()
>>> auto[0][10][100] = None
>>> assert str(auto) == '{0: {10: {100: None}}}'
```

to_dict()

Recursively casts a AutoDict into a regular dictionary. All nested AutoDict values are also converted.

Returns a copy of this dict without autovivification

Return type dict

Example

```
>>> from ubelt.util_dict import AutoDict
>>> auto = AutoDict()
>>> auto[1] = 1
>>> auto['n1'] = AutoDict()
>>> static = auto.to_dict()
>>> assert not isinstance(static, AutoDict)
>>> assert not isinstance(static['n1'], AutoDict)
```

class ubelt.util_dict.**AutoOrderedDict**

Bases: `collections.OrderedDict`, *`ubelt.util_dict.AutoDict`*

An infinitely nested default dict of dicts that maintains the ordering of items.

SeeAlso: *`AutoDict`* - the unordered version of this class

Example

```
>>> import ubelt as ub
>>> auto = ub.AutoOrderedDict()
>>> auto[0][3] = 3
>>> auto[0][2] = 2
>>> auto[0][1] = 1
>>> assert list(auto[0].values()) == [3, 2, 1]
```

ubelt.util_dict.**dzip**(*items1*, *items2*, *cls=<class 'dict'>*)

Zips elementwise pairs between items1 and items2 into a dictionary.

Values from items2 can be broadcast onto items1.

Parameters

- **items1** (*Iterable[KT]*) – full sequence
- **items2** (*Iterable[VT]*) – can either be a sequence of one item or a sequence of equal length to `items1`
- **cls** (*Type[dict], default=dict*) – dictionary type to use.

Returns similar to `dict(zip(items1, items2))`.

Return type Dict[KT, VT]

Example

```
>>> import ubelt as ub
>>> assert ub.dzip([1, 2, 3], [4]) == {1: 4, 2: 4, 3: 4}
>>> assert ub.dzip([1, 2, 3], [4, 4, 4]) == {1: 4, 2: 4, 3: 4}
>>> assert ub.dzip([], [4]) == {}
```

`ubelt.util_dict.`**`ddict`**

alias of `collections.defaultdict`

`ubelt.util_dict.`**`dict_hist`**(*items*, *weights=None*, *ordered=False*, *labels=None*)

Builds a histogram of items, counting the number of time each item appears in the input.

Parameters

- **items** (*Iterable[T]*) – hashable items (usually containing duplicates)
- **weights** (*Iterable[float], default=None*) – Corresponding weights for each item.
- **ordered** (*bool, default=False*) – If True the result is ordered by frequency.
- **labels** (*Iterable[T], default=None*) – Expected labels. Allows this function to pre-initialize the histogram. If specified the frequency of each label is initialized to zero and `items` can only contain items specified in labels.

Returns dictionary where the keys are unique elements from `items`, and the values are the number of times the item appears in `items`.

Return type dict[T, int]

Example

```
>>> import ubelt as ub
>>> items = [1, 2, 39, 900, 1232, 900, 1232, 2, 2, 2, 900]
>>> hist = ub.dict_hist(items)
>>> print(ub.repr2(hist, nl=0))
{1: 1, 2: 4, 39: 1, 900: 3, 1232: 2}
```

Example

```
>>> import ubelt as ub
>>> items = [1, 2, 39, 900, 1232, 900, 1232, 2, 2, 2, 900]
>>> hist1 = ub.dict_hist(items)
>>> hist2 = ub.dict_hist(items, ordered=True)
>>> try:
>>>     hist3 = ub.dict_hist(items, labels=[])
>>> except KeyError:
>>>     pass
>>> else:
>>>     raise AssertionError('expected key error')
```

(continues on next page)

(continued from previous page)

```
>>> weights = [1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1]
>>> hist4 = ub.dict_hist(items, weights=weights)
>>> print(ub.repr2(hist1, nl=0))
{1: 1, 2: 4, 39: 1, 900: 3, 1232: 2}
>>> print(ub.repr2(hist4, nl=0))
{1: 1, 2: 4, 39: 1, 900: 1, 1232: 0}
```

`ubelt.util_dict.`**`dict_subset`**(*dict_*, *keys*, *default=NoParam*, *cls=<class 'collections.OrderedDict'>*)

Get a subset of a dictionary

Parameters

- **dict_** (*Dict[KT, VT]*) – superset dictionary
- **keys** (*Iterable[KT]*) – keys to take from `dict_`
- **default** (*Optional[object] | util_const._NoParamType*) – if specified uses default if keys are missing.
- **cls** (*Type[Dict], default=OrderedDict*) – type of the returned dictionary.

Returns subset dictionary

Return type Dict[KT, VT]

SeeAlso: *`dict_isect()`* - similar functionality, but ignores missing keys

Example

```
>>> import ubelt as ub
>>> dict_ = {'K': 3, 'dcvs_clip_max': 0.2, 'p': 0.1}
>>> keys = ['K', 'dcvs_clip_max']
>>> subdict_ = ub.dict_subset(dict_, keys)
>>> print(ub.repr2(subdict_, nl=0))
{'K': 3, 'dcvs_clip_max': 0.2}
```

`ubelt.util_dict.`**`dict_union`**(**args*)

Dictionary set extension for `set.union`

Combines items with from multiple dictionaries. For items with intersecting keys, dictionaries towards the end of the sequence are given precedence.

Parameters ***args** (*List[Dict]*) – A sequence of dictionaries. Values are taken from the last

Returns OrderedDict if the first argument is an OrderedDict, otherwise dict

Return type Dict | OrderedDict

Notes

In Python 3.8+, the bitwise or operator "|" operator performs a similar operation, but as of 2022-06-01 there is still no public method for dictionary union (or any other dictionary set operator).

References

https://stackoverflow.com/questions/38987/merge-two-dict

SeeAlso: `collections.ChainMap()` - a standard python builtin data structure that provides a view that treats multiple dicts as a single dict. https://docs.python.org/3/library/collections.html#chainmap-objects

Example

```
>>> import ubelt as ub
>>> result = ub.dict_union({'a': 1, 'b': 1}, {'b': 2, 'c': 2})
>>> assert result == {'a': 1, 'b': 2, 'c': 2}
>>> ub.dict_union(
>>>     ub.odict([('a', 1), ('b', 2)]),
>>>     ub.odict([('c', 3), ('d', 4)]))
OrderedDict([('a', 1), ('b', 2), ('c', 3), ('d', 4)])
>>> ub.dict_union()
{}
```

`ubelt.util_dict.`**`dict_isect`**(**args*)

Dictionary set extension for `set.intersection()`

Constructs a dictionary that contains keys common between all inputs. The returned values will only belong to the first dictionary.

Parameters ***args** (*List[Dict[KT, VT] | Iterable[KT]]*) – A sequence of dictionaries (or sets of keys). The first argument should always be a dictionary, but the subsequent arguments can just be sets of keys.

Returns OrderedDict if the first argument is an OrderedDict, otherwise dict

Return type Dict[KT, VT] | OrderedDict[KT, VT]

Note: This function can be used as an alternative to *`dict_subset()`* where any key not in the dictionary is ignored. See the following example:

```
>>> import ubelt as ub
>>> # xdoctest: +IGNORE_WANT
>>> ub.dict_isect({'a': 1, 'b': 2, 'c': 3}, ['a', 'c', 'd'])
{'a': 1, 'c': 3}
```

Example

```
>>> import ubelt as ub
>>> ub.dict_isect({'a': 1, 'b': 1}, {'b': 2, 'c': 2})
{'b': 1}
>>> ub.dict_isect(odict([('a', 1), ('b', 2)]), odict([('c', 3)]))
OrderedDict()
>>> ub.dict_isect()
{}
```

ubelt.util_dict.**dict_diff**(**args*)

Dictionary set extension for `set.difference()`

Constructs a dictionary that contains any of the keys in the first arg, which are not in any of the following args.

Parameters ***args** (*List[Dict[KT, VT] | Iterable[KT]]*) – A sequence of dictionaries (or sets of keys). The first argument should always be a dictionary, but the subsequent arguments can just be sets of keys.

Returns OrderedDict if the first argument is an OrderedDict, otherwise dict

Return type Dict[KT, VT] | OrderedDict[KT, VT]

Todo:

- [ ] Add inplace keyword argument, which modifies the first dictionary inplace.

Example

```
>>> import ubelt as ub
>>> ub.dict_diff({'a': 1, 'b': 1}, {'a'}, {'c'})
{'b': 1}
>>> ub.dict_diff(odict([('a', 1), ('b', 2)]), odict([('c', 3)]))
OrderedDict([('a', 1), ('b', 2)])
>>> ub.dict_diff()
{}
>>> ub.dict_diff({'a': 1, 'b': 2}, {'c'})
```

ubelt.util_dict.**find_duplicates**(*items*, *k=2*, *key=None*)

Find all duplicate items in a list.

Search for all items that appear more than **k** times and return a mapping from each (k)-duplicate item to the positions it appeared in.

Parameters

- **items** (*Iterable[T]*) – hashable items possibly containing duplicates
- **k** (*int, default=2*) – only return items that appear at least **k** times.
- **key** (*Callable[[T], Any], default=None*) – Returns indices where *key(items[i])* maps to a particular value at least k times.

Returns maps each duplicate item to the indices at which it appears

Return type dict[T, List[int]]

Notes

Similar to `more_itertools.duplicates_everseen()`, `more_itertools.duplicates_justseen()`.

Example

```
>>> import ubelt as ub
>>> items = [0, 0, 1, 2, 3, 3, 0, 12, 2, 9]
>>> duplicates = ub.find_duplicates(items)
>>> # Duplicates are a mapping from each item that occurs 2 or more
>>> # times to the indices at which they occur.
>>> assert duplicates == {0: [0, 1, 6], 2: [3, 8], 3: [4, 5]}
>>> # You can set k=3 if you want to don't mind duplicates but you
>>> # want to find triplicates or quadruplets etc.
>>> assert ub.find_duplicates(items, k=3) == {0: [0, 1, 6]}
```

Example

```
>>> import ubelt as ub
>>> items = [0, 0, 1, 2, 3, 3, 0, 12, 2, 9]
>>> # note: k can less then 2
>>> duplicates = ub.find_duplicates(items, k=0)
>>> print(ub.repr2(duplicates, nl=0))
{0: [0, 1, 6], 1: [2], 2: [3, 8], 3: [4, 5], 9: [9], 12: [7]}
```

Example

```
>>> import ubelt as ub
>>> items = [10, 11, 12, 13, 14, 15, 16]
>>> duplicates = ub.find_duplicates(items, key=lambda x: x // 2)
>>> print(ub.repr2(duplicates, nl=0))
{5: [0, 1], 6: [2, 3], 7: [4, 5]}
```

`ubelt.util_dict.`**`group_items`**(*items*, *key*)

Groups a list of items by group id.

Parameters

- **items** (*Iterable[VT]*) – a list of items to group
- **key** (*Iterable[KT] | Callable[[VT], KT]*) – either a corresponding list of group-ids for each item or a function used to map each item to a group-id.

Returns a mapping from each group id to the list of corresponding items

Return type dict[KT, List[VT]]

Example

```
>>> import ubelt as ub
>>> items    = ['ham',     'jam',   'spam',     'eggs',    'cheese', 'banana']
>>> groupids = ['protein', 'fruit', 'protein',  'protein', 'dairy',  'fruit']
>>> id_to_items = ub.group_items(items, groupids)
>>> print(ub.repr2(id_to_items, nl=0))
{'dairy': ['cheese'], 'fruit': ['jam', 'banana'], 'protein': ['ham', 'spam', 'eggs
↪']}
```

`ubelt.util_dict.`**`invert_dict`**(*dict_*, *unique_vals=True*)

Swaps the keys and values in a dictionary.

Parameters

- **dict_** (*Dict[KT, VT]*) – dictionary to invert
- **unique_vals** (*bool, default=True*) – if False, the values of the new dictionary are sets of the original keys.

Returns the inverted dictionary

Return type Dict[VT, KT] | Dict[VT, Set[KT]]

Note: The must values be hashable.

If the original dictionary contains duplicate values, then only one of the corresponding keys will be returned and the others will be discarded. This can be prevented by setting `unique_vals=False`, causing the inverted keys to be returned in a set.

Example

```
>>> import ubelt as ub
>>> dict_ = {'a': 1, 'b': 2}
>>> inverted = ub.invert_dict(dict_)
>>> assert inverted == {1: 'a', 2: 'b'}
```

Example

```
>>> import ubelt as ub
>>> dict_ = ub.odict([(2, 'a'), (1, 'b'), (0, 'c'), (None, 'd')])
>>> inverted = ub.invert_dict(dict_)
>>> assert list(inverted.keys())[0] == 'a'
```

Example

```
>>> import ubelt as ub
>>> dict_ = {'a': 1, 'b': 0, 'c': 0, 'd': 0, 'f': 2}
>>> inverted = ub.invert_dict(dict_, unique_vals=False)
>>> assert inverted == {0: {'b', 'c', 'd'}, 1: {'a'}, 2: {'f'}}
```

ubelt.util_dict.**map_keys**(*func*, *dict_*)

Apply a function to every key in a dictionary.

Creates a new dictionary with the same values and modified keys. An error is raised if the new keys are not unique.

Parameters

- **func** (*Callable[[KT], T] | Mapping[KT, T]*) – a function or indexable object
- **dict_** (*Dict[KT, VT]*) – a dictionary

Returns transformed dictionary

Return type Dict[T, VT]

Raises `Exception` – if multiple keys map to the same value

Example

```
>>> import ubelt as ub
>>> dict_ = {'a': [1, 2, 3], 'b': []}
>>> func = ord
>>> newdict = ub.map_keys(func, dict_)
>>> print(newdict)
>>> assert newdict == {97: [1, 2, 3], 98: []}
>>> dict_ = {0: [1, 2, 3], 1: []}
>>> func = ['a', 'b']
>>> newdict = ub.map_keys(func, dict_)
>>> print(newdict)
>>> assert newdict == {'a': [1, 2, 3], 'b': []}
```

ubelt.util_dict.**map_vals**(*func*, *dict_*)

Apply a function to every value in a dictionary.

Creates a new dictionary with the same keys and modified values.

Parameters

- **func** (*Callable[[VT], T] | Mapping[VT, T]*) – a function or indexable object
- **dict_** (*Dict[KT, VT]*) – a dictionary

Returns transformed dictionary

Return type Dict[KT, T]

Example

```
>>> import ubelt as ub
>>> dict_ = {'a': [1, 2, 3], 'b': []}
>>> newdict = ub.map_vals(len, dict_)
>>> assert newdict ==  {'a': 3, 'b': 0}
```

Example

```
>>> # Can also use an indexable as ``func``
>>> import ubelt as ub
>>> dict_ = {'a': 0, 'b': 1}
>>> func = [42, 21]
>>> newdict = ub.map_vals(func, dict_)
>>> assert newdict ==  {'a': 42, 'b': 21}
>>> print(newdict)
```

`ubelt.util_dict.`**`sorted_keys`**(*dict_*, *key=None*, *reverse=False*)

Return an ordered dictionary sorted by its keys

Parameters

- **dict_** (*Dict[KT, VT]*) – dictionary to sort. The keys must be of comparable types.
- **key** (*Callable[[KT], Any] | None*) – If given as a callable, customizes the sorting by ordering using transformed keys.
- **reverse** (*bool, default=False*) – if True returns in descending order

Returns new dictionary where the keys are ordered

Return type OrderedDict[KT, VT]

Example

```
>>> import ubelt as ub
>>> dict_ = {'spam': 2.62, 'eggs': 1.20, 'jam': 2.92}
>>> newdict = sorted_keys(dict_)
>>> print(ub.repr2(newdict, nl=0))
{'eggs': 1.2, 'jam': 2.92, 'spam': 2.62}
>>> newdict = sorted_keys(dict_, reverse=True)
>>> print(ub.repr2(newdict, nl=0))
{'spam': 2.62, 'jam': 2.92, 'eggs': 1.2}
>>> newdict = sorted_keys(dict_, key=lambda x: sum(map(ord, x)))
>>> print(ub.repr2(newdict, nl=0))
{'jam': 2.92, 'eggs': 1.2, 'spam': 2.62}
```

`ubelt.util_dict.`**`sorted_vals`**(*dict_*, *key=None*, *reverse=False*)

Return an ordered dictionary sorted by its values

Parameters

- **dict_** (*Dict[KT, VT]*) – dictionary to sort. The values must be of comparable types.

- **key** (*Callable[[VT], Any] | None*) – If given as a callable, customizes the sorting by ordering using transformed values.
- **reverse** (*bool, default=False*) – if True returns in descending order

Returns new dictionary where the values are ordered

Return type OrderedDict[KT, VT]

Example

```
>>> import ubelt as ub
>>> dict_ = {'spam': 2.62, 'eggs': 1.20, 'jam': 2.92}
>>> newdict = sorted_vals(dict_)
>>> print(ub.repr2(newdict, nl=0))
{'eggs': 1.2, 'spam': 2.62, 'jam': 2.92}
>>> newdict = sorted_vals(dict_, reverse=True)
>>> print(ub.repr2(newdict, nl=0))
{'jam': 2.92, 'spam': 2.62, 'eggs': 1.2}
>>> newdict = sorted_vals(dict_, key=lambda x: x % 1.6)
>>> print(ub.repr2(newdict, nl=0))
{'spam': 2.62, 'eggs': 1.2, 'jam': 2.92}
```

`ubelt.util_dict.`**`odict`**

alias of `collections.OrderedDict`

`ubelt.util_dict.`**`named_product`**`(_=None, **basis)`

Generates the Cartesian product of the `basis.values()`, where each generated item labeled by `basis.keys()`.

In other words, given a dictionary that maps each "axes" (i.e. some variable) to its "basis" (i.e. the possible values that it can take), generate all possible points in that grid (i.e. unique assignments of variables to values).

Parameters

- _ (*Dict[str, List[VT]] | None, default=None*) – Use of this positional argument is not recommend. Instead specify all arguments as keyword args.

  If specified, this should be a dictionary is unioned with the keyword args. This exists to support ordered dictionaries before Python 3.6, and may eventually be removed.

- **basis** (*Dict[str, List[VT]]*) – A dictionary where the keys correspond to "columns" and the values are a list of possible values that "column" can take.

  I.E. each key corresponds to an "axes", the values are the list of possible values for that "axes".

Yields *Dict[str, VT]* – a "row" in the "longform" data containing a point in the Cartesian product.

Note: This function is similar to `itertools.product()`, the only difference is that the generated items are a dictionary that retains the input keys instead of an tuple.

This function used to be called "basis_product", but "named_product" might be more appropriate. This function exists in other places ([minstrel271_namedproduct], [pytb_namedproduct], and [Hettinger_namedproduct]).

References

Example

```
>>> # An example use case is looping over all possible settings in a
>>> # configuration dictionary for a grid search over parameters.
>>> import ubelt as ub
>>> basis = {
>>>     'arg1': [1, 2, 3],
>>>     'arg2': ['A1', 'B1'],
>>>     'arg3': [9999, 'Z2'],
>>>     'arg4': ['always'],
>>> }
>>> import ubelt as ub
>>> # sort input data for older python versions
>>> basis = ub.odict(sorted(basis.items()))
>>> got = list(ub.named_product(basis))
>>> print(ub.repr2(got, nl=-1))
[
    {'arg1': 1, 'arg2': 'A1', 'arg3': 9999, 'arg4': 'always'},
    {'arg1': 1, 'arg2': 'A1', 'arg3': 'Z2', 'arg4': 'always'},
    {'arg1': 1, 'arg2': 'B1', 'arg3': 9999, 'arg4': 'always'},
    {'arg1': 1, 'arg2': 'B1', 'arg3': 'Z2', 'arg4': 'always'},
    {'arg1': 2, 'arg2': 'A1', 'arg3': 9999, 'arg4': 'always'},
    {'arg1': 2, 'arg2': 'A1', 'arg3': 'Z2', 'arg4': 'always'},
    {'arg1': 2, 'arg2': 'B1', 'arg3': 9999, 'arg4': 'always'},
    {'arg1': 2, 'arg2': 'B1', 'arg3': 'Z2', 'arg4': 'always'},
    {'arg1': 3, 'arg2': 'A1', 'arg3': 9999, 'arg4': 'always'},
    {'arg1': 3, 'arg2': 'A1', 'arg3': 'Z2', 'arg4': 'always'},
    {'arg1': 3, 'arg2': 'B1', 'arg3': 9999, 'arg4': 'always'},
    {'arg1': 3, 'arg2': 'B1', 'arg3': 'Z2', 'arg4': 'always'}
]
```

Example

```
>>> import ubelt as ub
>>> list(ub.named_product(a=[1, 2, 3]))
[{'a': 1}, {'a': 2}, {'a': 3}]
>>> # xdoctest: +IGNORE_WANT
>>> list(ub.named_product(a=[1, 2, 3], b=[4, 5]))
[{'a': 1, 'b': 4},
 {'a': 1, 'b': 5},
 {'a': 2, 'b': 4},
 {'a': 2, 'b': 5},
 {'a': 3, 'b': 4},
 {'a': 3, 'b': 5}]
```

`ubelt.util_dict.`**`varied_values`**(*longform*, *min_variations=0*, *default=NoParam*)

Given a list of dictionaries, find the values that differ between them.

Parameters

- **longform** (*List[Dict[KT, VT]]*) – This is longform data, as described in [SeabornLongform]. It is a list of dictionaries.

  Each item in the list - or row - is a dictionary and can be thought of as an observation. The keys in each dictionary are the columns. The values of the dictionary must be hashable. Lists will be converted into tuples.

- **min_variations** (*int, default=0*) – "columns" with fewer than `min_variations` unique values are removed from the result.

- **default** (*VT, default=NoParam*) – if specified, unspecified columns are given this value.

Returns a mapping from each "column" to the set of unique values it took over each "row". If a column is not specified for each row, it is assumed to take a *default* value, if it is specified.

Return type Dict[KT, List[VT]]

Raises `KeyError` – If `default` is unspecified and all the rows do not contain the same columns.

References

Example

```
>>> # An example use case is to determine what values of a
>>> # configuration dictionary were tried in a random search
>>> # over a parameter grid.
>>> import ubelt as ub
>>> longform = [
>>>     {'col1': 1, 'col2': 'foo', 'col3': None},
>>>     {'col1': 1, 'col2': 'foo', 'col3': None},
>>>     {'col1': 2, 'col2': 'bar', 'col3': None},
>>>     {'col1': 3, 'col2': 'bar', 'col3': None},
>>>     {'col1': 9, 'col2': 'bar', 'col3': None},
>>>     {'col1': 1, 'col2': 'bar', 'col3': None},
>>> ]
>>> varied = ub.varied_values(longform)
>>> print('varied = {}'.format(ub.repr2(varied, nl=1)))
varied = {
    'col1': {1, 2, 3, 9},
    'col2': {'bar', 'foo'},
    'col3': {None},
}
```

Example

```
>>> import ubelt as ub
>>> import random
>>> longform = [
>>>     {'col1': 1, 'col2': 'foo', 'col3': None},
>>>     {'col1': 1, 'col2': [1, 2], 'col3': None},
>>>     {'col1': 2, 'col2': 'bar', 'col3': None},
>>>     {'col1': 3, 'col2': 'bar', 'col3': None},
>>>     {'col1': 9, 'col2': 'bar', 'col3': None},
>>>     {'col1': 1, 'col2': 'bar', 'col3': None, 'extra_col': 3},
```

(continues on next page)

(continued from previous page)

```
>>> ]
>>> # Operation fails without a default
>>> import pytest
>>> with pytest.raises(KeyError):
>>>     varied = ub.varied_values(longform)
>>> #
>>> # Operation works with a default
>>> varied = ub.varied_values(longform, default='<unset>')
>>> expected = {
>>>     'col1': {1, 2, 3, 9},
>>>     'col2': {'bar', 'foo', (1, 2)},
>>>     'col3': set([None]),
>>>     'extra_col': {'<unset>', 3},
>>> }
>>> print('varied = {!r}'.format(varied))
>>> assert varied == expected
```

Example

```
>>> # xdoctest: +REQUIRES(PY3)
>>> # Random numbers are different in Python2, so skip in that case
>>> import ubelt as ub
>>> import random
>>> num_cols = 11
>>> num_rows = 17
>>> rng = random.Random(0)
>>> # Generate a set of columns
>>> columns = sorted(ub.hash_data(i)[0:8] for i in range(num_cols))
>>> # Generate rows for each column
>>> longform = [
>>>     {key: ub.hash_data(key)[0:8] for key in columns}
>>>     for _ in range(num_rows)
>>> ]
>>> # Add in some varied values in random positions
>>> for row in longform:
>>>     if rng.random() > 0.5:
>>>         for key in sorted(row.keys()):
>>>             if rng.random() > 0.95:
>>>                 row[key] = 'special-' + str(rng.randint(1, 32))
>>> varied = ub.varied_values(longform, min_variations=1)
>>> print('varied = {}'.format(ub.repr2(varied, nl=1, sort=True)))
varied = {
    '095f3e44': {'8fb4d4c9', 'special-23'},
    '365d11a1': {'daa409da', 'special-31', 'special-32'},
    '5815087d': {'1b823610', 'special-3'},
    '7b54b668': {'349a782c', 'special-10'},
    'b8244d02': {'d57bca90', 'special-8'},
    'f27b5bf8': {'fa0f90d1', 'special-19'},
}
```

1.27.1.1.1.9 ubelt.util_download module

Helpers for downloading data

The `download()` function access the network and requests the content at a specific url using `urllib` or `urllib2`. You can either specify where the data goes or download it to the default location in ubelt cache. Either way this function returns the location of the downloaded data. You can also specify the expected hash in order to check the validity of the data. By default downloading is verbose.

The `grabdata()` function is almost identitcal to `download()`, but it checks if the data already exists in the download location, and only downloads if it needs to.

`ubelt.util_download.`**`download`**(*url*, *fpath=None*, *dpath=None*, *fname=None*, *hash_prefix=None*, *hasher='sha512'*, *chunksize=8192*, *verbose=1*, *timeout=NoParam*, *progkw=None*)

Downloads a url to a file on disk.

If unspecified the location and name of the file is chosen automatically. A hash_prefix can be specified to verify the integrity of the downloaded data. This function will download the data every time its called. For cached downloading see *grabdata*.

Parameters

- **url** (*str*) – The url to download.
- **fpath** (*Optional[str | PathLike | io.BytesIO]*) – The path to download to. Defaults to basename of url and ubelt's application cache. If this is a io.BytesIO object then information is directly written to this object (note this prevents the use of temporary files).
- **dpath** (*Optional[PathLike]*) – where to download the file. If unspecified *appname* is used to determine this. Mutually exclusive with fpath.
- **fname** (*Optional[str]*) – What to name the downloaded file. Defaults to the url basename. Mutually exclusive with fpath.
- **hash_prefix** (*None | str*) – If specified, download will retry / error if the file hash does not match this value. Defaults to None.
- **hasher** (*str | Hasher*) – If hash_prefix is specified, this indicates the hashing algorithm to apply to the file. Defaults to sha512.
- **chunksize** (*int, default=2 * 13**) – Download chunksize.
- **verbose** (*int, default=1*) – Verbosity level 0 or 1.
- **timeout** (*float, default=NoParam*) – Specify timeout in seconds for `urllib.request.urlopen()`. (if not specified, the global default timeout setting will be used) This only works for HTTP, HTTPS and FTP connections for blocking operations like the connection attempt.
- **progkw** (*Dict | None*) – if specified provides extra arguments to the progress iterator. See `ubelt.progiter.ProgIter` for available options.

Returns fpath - path to the downloaded file.

Return type str | PathLike

Raises

- **`URLError - if there is problem downloading the url`** –
- **`RuntimeError - if the hash does not match the hash_prefix`** –

Note: Based largely on code in pytorch [TorchDL] with modifications influenced by other resources [Shichao_2012] [SO_15644964] [SO_16694907].

References

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> from ubelt.util_download import *  # NOQA
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> fpath = download(url)
>>> print(basename(fpath))
rqwaDag.png
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import ubelt as ub
>>> import io
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> file = io.BytesIO()
>>> fpath = ub.download(url, file)
>>> file.seek(0)
>>> data = file.read()
>>> assert ub.hash_data(data, hasher='sha1').startswith('f79ea24571')
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> fpath = download(url, hasher='sha1', hash_prefix=
↪'f79ea24571da6ddd2ba12e3d57b515249ecb8a35')
Downloading url='http://i.imgur.com/rqwaDag.png' to fpath=...rqwaDag.png
...
...1233/1233... rate=... Hz, eta=..., total=...
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import pytest
>>> import ubelt as ub
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> #fpath = download(url, hasher='sha1', hash_prefix=
↪'f79ea24571da6ddd2ba12e3d57b515249ecb8a35')
>>> # test download from girder
>>> #url = 'https://data.kitware.com/api/v1/item/5b4039308d777f2e6225994c/download'
```

(continues on next page)

(continued from previous page)

```
>>> #ub.download(url, hasher='sha512', hash_prefix='c98a46cb31205cf')
>>> with pytest.raises(RuntimeError):
>>>     ub.download(url, hasher='sha512', hash_prefix='BAD_HASH')
```

`ubelt.util_download.grabdata`(*url*, *fpath=None*, *dpath=None*, *fname=None*, *redo=False*, *verbose=1*, *appname=None*, *hash_prefix=None*, *hasher='sha512'*, ***download_kw*)

Downloads a file, caches it, and returns its local path.

If unspecified the location and name of the file is chosen automatically. A hash_prefix can be specified to verify the integrity of the downloaded data.

Parameters

- **url** (*str*) – url of the file to download
- **fpath** (*Optional[str | PathLike]*) – The full path to download the file to. If unspecified, the arguments *dpath* and *fname* are used to determine this.
- **dpath** (*Optional[str | PathLike]*) – where to download the file. If unspecified *appname* is used to determine this. Mutually exclusive with fpath.
- **fname** (*Optional[str]*) – What to name the downloaded file. Defaults to the url basename. Mutually exclusive with fpath.
- **redo** (*bool, default=False*) – if True forces redownload of the file
- **verbose** (*bool, default=True*) – verbosity flag
- **appname** (*str*) – set dpath to *ub.get_app_cache_dir(appname)*. Mutually exclusive with dpath and fpath.
- **hash_prefix** (*None | str*) – If specified, grabdata verifies that this matches the hash of the file, and then saves the hash in a adjacent file to certify that the download was successful. Defaults to None.
- **hasher** (*str | Hasher*) – If hash_prefix is specified, this indicates the hashing algorithm to apply to the file. Defaults to sha512. NOTE: Only pass hasher as a string. Passing as an instance is deprecated and can cause unexpected results.
- ****download_kw** – additional kwargs to pass to `ubelt.util_download.download()`

Returns fpath - path to downloaded or cached file.

Return type str | PathLike

CommandLine

```
xdoctest -m ubelt.util_download grabdata --network
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import ubelt as ub
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> fpath = ub.grabdata(url, fname='mario.png')
>>> result = basename(fpath)
>>> print(result)
mario.png
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import ubelt as ub
>>> import json
>>> fname = 'foo.bar'
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> prefix1 = '944389a39dfb8fa9'
>>> fpath = ub.grabdata(url, fname=fname, hash_prefix=prefix1, verbose=3)
>>> stamp_fpath = ub.Path(fpath + '.stamp_sha512.json')
>>> assert json.loads(stamp_fpath.read_text())['hash'][0].startswith(prefix1)
>>> # Check that the download doesn't happen again
>>> fpath = ub.grabdata(url, fname=fname, hash_prefix=prefix1)
>>> # todo: check file timestamps have not changed
>>> #
>>> # Check redo works with hash
>>> fpath = ub.grabdata(url, fname=fname, hash_prefix=prefix1, redo=True)
>>> # todo: check file timestamps have changed
>>> #
>>> # Check that a redownload occurs when the stamp is changed
>>> with open(stamp_fpath, 'w') as file:
>>>     file.write('corrupt-stamp')
>>> fpath = ub.grabdata(url, fname=fname, hash_prefix=prefix1)
>>> assert json.loads(stamp_fpath.read_text())['hash'][0].startswith(prefix1)
>>> #
>>> # Check that a redownload occurs when the stamp is removed
>>> ub.delete(stamp_fpath)
>>> with open(fpath, 'w') as file:
>>>     file.write('corrupt-data')
>>> assert not ub.hash_file(fpath, base='hex', hasher='sha512').startswith(prefix1)
>>> fpath = ub.grabdata(url, fname=fname, hash_prefix=prefix1)
>>> assert ub.hash_file(fpath, base='hex', hasher='sha512').startswith(prefix1)
>>> #
>>> # Check that requesting new data causes redownload
>>> #url2 = 'https://data.kitware.com/api/v1/item/5b4039308d777f2e6225994c/download'
>>> #prefix2 = 'c98a46cb31205cf'  # hack SSL
>>> url2 = 'http://i.imgur.com/rqwaDag.png'
>>> prefix2 = '944389a39dfb8fa9'
>>> fpath = ub.grabdata(url2, fname=fname, hash_prefix=prefix2)
>>> assert json.loads(stamp_fpath.read_text())['hash'][0].startswith(prefix2)
```

1.27.1.1.1.10 ubelt.util_download_manager module

A simple download manager

class ubelt.util_download_manager.**DownloadManager**(*download_root=None*, *mode='thread'*, *max_workers=None*, *cache=True*)

Bases: object

Simple implementation of the download manager

Variables

- **download_root** (*str | PathLike*) – default download location
- **jobs** (*List[concurrent.futures.Future]*) – list of jobs

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import ubelt as ub
>>> # Download a file with a known hash
>>> manager = ub.DownloadManager()
>>> job = manager.submit(
>>>     'http://i.imgur.com/rqwaDag.png',
>>>     hash_prefix=
↪'31a129618c87dd667103e7154182e3c39a605eefe90f84f2283f3c87efee8e40'
>>> )
>>> fpath = job.result()
>>> print('fpath = {!r}'.format(fpath))
```

Example

```
>>> # Does not require network
>>> import ubelt as ub
>>> manager = ub.DownloadManager()
>>> for i in range(100):
...     job = manager.submit('localhost/might-not-exist-i-{}'.format(i))
>>> file_paths = []
>>> for job in manager.as_completed(prog=True):
...     try:
...         fpath = job.result()
...         file_paths += [fpath]
...     except Exception:
...         pass
>>> print('file_paths = {!r}'.format(file_paths))
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import pytest
>>> import ubelt as ub
>>> manager = ub.DownloadManager()
>>> item1 = {
>>>     'url': 'https://data.kitware.com/api/v1/item/5b4039308d777f2e6225994c/
↪download',
>>>     'dst': 'forgot_what_the_name_really_is',
>>>     'hash_prefix': 'c98a46cb31205cf',
>>>     'hasher': 'sha512',
>>> }
>>> item2 = {
>>>     'url': 'http://i.imgur.com/rqwaDag.png',
>>>     'hash_prefix': 'f79ea24571da6ddd2ba12e3d57b515249ecb8a35',
>>>     'hasher': 'sha1',
>>> }
>>> item1 = item2  # hack around SSL error
>>> manager.submit(**item1)
>>> manager.submit(**item2)
>>> for job in manager.as_completed(prog=True, verbose=3):
>>>     fpath = job.result()
>>>     print('fpath = {!r}'.format(fpath))
```

submit(*url*, *dst=None*, *hash_prefix=None*, *hasher='sha256'*)

Add a job to the download Queue

Parameters

- **url** (*str | PathLike*) – pointer to the data to download
- **dst** (*str | None*) – The relative or absolute path to download to. If unspecified, the destination name is derived from the url.
- **hash_prefix** (*str | None*) – If specified, verifies that the hash of the downloaded file starts with this.
- **hasher** (*str, default='sha256'*) – hashing algorithm to use if hash_prefix is specified.

Returns a Future object that will point to the downloaded location.

Return type concurrent.futures.Future

as_completed(*prog=None*, *desc=None*, *verbose=1*)

Generate completed jobs as they become available

Example

```
>>> import pytest
>>> import ubelt as ub
>>> download_root = ub.ensure_app_config_dir('ubelt', 'dlman')
>>> manager = ub.DownloadManager(download_root=download_root,
>>>                              cache=False)
>>> for i in range(3):
>>>     manager.submit('localhost')
>>> results = list(manager)
>>> print('results = {!r}'.format(results))
>>> manager.shutdown()
```

shutdown()

Cancel all jobs and close all connections.

1.27.1.1.1.11 ubelt.util_format module

Defines the function *repr2()*, which allows for a bit more customization than `repr()` or `pprint()`. See the docstring for more details.

Two main goals of repr2 are to provide nice string representations of nested data structures and make those "eval-able" whenever possible. As an example take the value `float('inf')`, which normally has a non-evalable repr of `inf`:

```
>>> import ubelt as ub
>>> ub.repr2(float('inf'))
"float('inf')"
```

The `newline` (or `nl`) keyword argument can control how deep in the nesting newlines are allowed.

```
>>> print(ub.repr2({1: float('nan'), 2: float('inf'), 3: 3.0}))
{
    1: float('nan'),
    2: float('inf'),
    3: 3.0,
}
```

```
>>> print(ub.repr2({1: float('nan'), 2: float('inf'), 3: 3.0}, nl=0))
{1: float('nan'), 2: float('inf'), 3: 3.0}
```

You can also define or overwrite how representations for different types are created. You can either create your own extension object, or you can monkey-patch *ub.util_format._FORMATTER_EXTENSIONS* without specifying the extensions keyword argument (although this will be a global change).

```
>>> extensions = ub.util_format.FormatterExtensions()
>>> @extensions.register(float)
>>> def my_float_formater(data, **kw):
>>>     return "monkey({})".format(data)
>>> print(ub.repr2({1: float('nan'), 2: float('inf'), 3: 3.0}, nl=0,␣
↪extensions=extensions))
{1: monkey(nan), 2: monkey(inf), 3: monkey(3.0)}
```

As of ubelt 1.1.0 you can now access and update the default extensions via the repr2 function itself.

```
>>> # xdoctest: +SKIP
>>> # We skip this at test time to not modify global state
>>> @ub.repr2.EXTENSIONS.register(float)
>>> def my_float_formater(data, **kw):
>>>     return "monkey2({})".format(data)
>>> print(ub.repr2({1: float('nan'), 2: float('inf'), 3: 3.0}, nl=0))
```

ubelt.util_format.**repr2**(*data*, ***kwargs*)

Makes a pretty string representation of `data`.

Makes a pretty and easy-to-doctest string representation. Has nice handling of common nested datatypes. This is an alternative to repr, and `pprint.pformat()`.

This output of this function are configurable. By default it aims to produce strings that are consistent, compact, and executable. This makes them great for doctests.

Note: This function has many keyword arguments that can be used to customize the final representation. For convenience some of the more frequently used kwargs have short aliases. See "Kwargs" for more details.

Parameters **data** (*object*) – an arbitrary python object to form the string "representation" of

Kwargs:

si, stritems, (bool): dict/list items use str instead of repr

strkeys, sk (bool): dict keys use str instead of repr

strvals, sv (bool): dict values use str instead of repr

nl, newlines (int | bool): number of top level nestings to place a newline after. If true all items are followed by newlines regardless of nesting level. Defaults to 1 for lists and True for dicts.

nobr, nobraces (bool, default=False): if True, text will not contain outer braces for containers

cbr, compact_brace (bool, default=False): if True, braces are compactified (i.e. they will not have newlines placed directly after them, think java / K&R / 1TBS)

trailsep, trailing_sep (bool): if True, a separator is placed after the last item in a sequence. By default this is True if there are any `nl > 0`.

explicit (bool, default=False): changes dict representation from `{k1: v1, ...}` to `dict(k1=v1, ...)`.

Modifies: default kvsep is modified to `'='` dict braces from *{}* to *dict()*.

compact (bool, default=False): Produces values more suitable for space constrianed environments

Modifies: default kvsep is modified to `'='` default itemsep is modified to `''` default nobraces is modified to `1`. default nl is modified to `0`. default strkeys to `True` default strvals to `True`

precision (int, default=None): if specified floats are formatted with this precision

kvsep (str, default=': '): separator between keys and values

itemsep (str, default=' '): separator between items

sort (bool | callable, default=None): if None, then sort unordered collections, but keep the ordering of ordered collections. This option attempts to be deterministic in most cases.

New in 0.8.0: if `sort` is callable, it will be used as a key-function to sort all collections.

if False, then nothing will be sorted, and the representation of unordered collections will be arbitrary and possibly non-determenistic.

if True, attempts to sort all collections in the returned text. Currently if True this WILL sort lists. Currently if True this WILL NOT sort OrderedDicts.

NOTE: The previous behavior may not be intuitive, as such the behavior of this arg is subject to change.

suppress_small (bool): passed to `numpy.array2string()` for ndarrays

max_line_width (int): passed to `numpy.array2string()` for ndarrays

with_dtype (bool): only relevant to numpy.ndarrays. if True includes the dtype. Defaults to *not strvals*.

align (bool | str, default=False): if True, will align multi-line dictionaries by the kvsep

extensions (FormatterExtensions): a custom *`FormatterExtensions`* instance that can overwrite or define how different types of objects are formatted.

Returns outstr - output string

Return type str

Note: There are also internal kwargs, which should not be used:

_return_info (bool): return information about child context

_root_info (depth): information about parent context

RelatedWork: `rich.pretty.pretty_repr()` `pprint.pformat()`

Example

```
>>> import ubelt as ub
>>> dict_ = {
...     'custom_types': [slice(0, 1, None), 1/3],
...     'nest_dict': {'k1': [1, 2, {3: {4, 5}}],
...                   'key2': [1, 2, {3: {4, 5}}],
...                   'key3': [1, 2, {3: {4, 5}}],
...                   },
...     'nest_dict2': {'k': [1, 2, {3: {4, 5}}]},
...     'nested_tuples': [tuple([1]), tuple([2, 3]), frozenset([4, 5, 6])],
...     'one_tup': tuple([1]),
...     'simple_dict': {'spam': 'eggs', 'ham': 'jam'},
...     'simple_list': [1, 2, 'red', 'blue'],
...     'odict': ub.odict([(1, '1'), (2, '2')]),
... }
>>> # In the interest of saving space we are only going to show the
>>> # output for the first example.
>>> result = ub.repr2(dict_, nl=1, precision=2)
>>> print(result)
{
    'custom_types': [slice(0, 1, None), 0.33],
    'nest_dict': {'k1': [1, 2, {3: {4, 5}}], 'key2': [1, 2, {3: {4, 5}}], 'key3':␣
↪[1, 2, {3: {4, 5}}]},
```

(continues on next page)

(continued from previous page)

```
    'nest_dict2': {'k': [1, 2, {3: {4, 5}}]},
    'nested_tuples': [(1,), (2, 3), {4, 5, 6}],
    'odict': {1: '1', 2: '2'},
    'one_tup': (1,),
    'simple_dict': {'ham': 'jam', 'spam': 'eggs'},
    'simple_list': [1, 2, 'red', 'blue'],
}
>>> # You can try the rest yourself.
>>> result = ub.repr2(dict_, nl=3, precision=2); print(result)
>>> result = ub.repr2(dict_, nl=2, precision=2); print(result)
>>> result = ub.repr2(dict_, nl=1, precision=2, itemsep='', explicit=True);␣
↪print(result)
>>> result = ub.repr2(dict_, nl=1, precision=2, nobr=1, itemsep='', explicit=True);␣
↪print(result)
>>> result = ub.repr2(dict_, nl=3, precision=2, cbr=True); print(result)
>>> result = ub.repr2(dict_, nl=3, precision=2, si=True); print(result)
>>> result = ub.repr2(dict_, nl=3, sort=True); print(result)
>>> result = ub.repr2(dict_, nl=3, sort=False, trailing_sep=False); print(result)
>>> result = ub.repr2(dict_, nl=3, sort=False, trailing_sep=False, nobr=True);␣
↪print(result)
```

Example

```
>>> import ubelt as ub
>>> def _nest(d, w):
...     if d == 0:
...         return {}
...     else:
...         return {'n{}'.format(d): _nest(d - 1, w + 1), 'm{}'.format(d): _nest(d -
↪ 1, w + 1)}
>>> dict_ = _nest(d=4, w=1)
>>> result = ub.repr2(dict_, nl=6, precision=2, cbr=1)
>>> print('---')
>>> print(result)
>>> result = ub.repr2(dict_, nl=-1, precision=2)
>>> print('---')
>>> print(result)
```

Example

```
>>> import ubelt as ub
>>> data = {'a': 100, 'b': [1, '2', 3], 'c': {20:30, 40: 'five'}}
>>> print(ub.repr2(data, nl=1))
{
    'a': 100,
    'b': [1, '2', 3],
    'c': {20: 30, 40: 'five'},
}
>>> # Compact is useful for things like timerit.Timerit labels
```

(continues on next page)

(continued from previous page)

```
>>> print(ub.repr2(data, compact=True))
a=100,b=[1,2,3],c={20=30,40=five}
>>> print(ub.repr2(data, compact=True, nobr=False))
{a=100,b=[1,2,3],c={20=30,40=five}}
```

class ubelt.util_format.**FormatterExtensions**

Bases: object

Helper class for managing non-builtin (e.g. numpy) format types.

This module (*ubelt.util_format*) maintains a global set of basic extensions, but it is also possible to create a locally scoped set of extensions and explicitly pass it to repr2. The following example demonstrates this.

Example

```
>>> import ubelt as ub
>>> class MyObject(object):
>>>     pass
>>> data = {'a': [1, 2.2222, MyObject()], 'b': MyObject()}
>>> # Create a custom set of extensions
>>> extensions = ub.FormatterExtensions()
>>> # Register a function to format your specific type
>>> @extensions.register(MyObject)
>>> def format_myobject(data, **kwargs):
>>>     return 'I can do anything here'
>>> # Repr2 will now respect the passed custom extensions
>>> # Note that the global extensions will still be respected
>>> # unless they are overloaded.
>>> print(ub.repr2(data, nl=-1, precision=1, extensions=extensions))
{
    'a': [1, 2.2, I can do anything here],
    'b': I can do anything here
}
>>> # Overload the formatter for float and int
>>> @extensions.register((float, int))
>>> def format_myobject(data, **kwargs):
>>>     return str((data + 10) // 2)
>>> print(ub.repr2(data, nl=-1, precision=1, extensions=extensions))
{
    'a': [5, 6.0, I can do anything here],
    'b': I can do anything here
}
```

register(*key*)

Registers a custom formatting function with ub.repr2

Parameters **key** (*Type | Tuple[Type] | str*) – indicator of the type

Returns decorator function

Return type Callable

lookup(*data*)

Returns an appropriate function to format `data` if one has been registered.

1.27.1.1.1.12 ubelt.util_func module

Helpers for functional programming.

The `identity()` function simply returns its own inputs. This is useful for bypassing print statements and many other cases. I also think it looks a little nicer than `lambda x: x`.

The `inject_method()` function "injects" another function into a class instance as a method. This is useful for monkey patching.

`ubelt.util_func.identity`(*arg=None*, **args*, ***kwargs*)

The identity function. Simply returns the value of its first input.

All other inputs are ignored. Defaults to None if called without args.

Parameters

- **arg** (*object, default=None*) – some value
- ***args** – ignored
- ****kwargs** – ignored

Returns arg - the same value

Return type object

Example

```
>>> import ubelt as ub
>>> ub.identity(42)
42
>>> ub.identity(42, 42)
42
>>> ub.identity()
None
```

`ubelt.util_func.inject_method`(*self*, *func*, *name=None*)

Injects a function into an object instance as a bound method

The main use case of this function is for monkey patching. While monkey patching is sometimes necessary it should generally be avoided. Thus, we simply remind the developer that there might be a better way.

Parameters

- **self** (*T*) – Instance to inject a function into.
- **func** (*Callable[..., Any]*) – The function to inject (must contain an arg for self).
- **name** (*str, default=None*) – Name of the method. optional. If not specified the name of the function is used.

Example

```
>>> import ubelt as ub
>>> class Foo(object):
>>>     def bar(self):
>>>         return 'bar'
>>> def baz(self):
>>>     return 'baz'
>>> self = Foo()
>>> assert self.bar() == 'bar'
>>> assert not hasattr(self, 'baz')
>>> ub.inject_method(self, baz)
>>> assert not hasattr(Foo, 'baz'), 'should only change one instance'
>>> assert self.baz() == 'baz'
>>> ub.inject_method(self, baz, 'bar')
>>> assert self.bar() == 'baz'
```

`ubelt.util_func.`**`compatible`**(*config*, *func*, *start=0*)

Take the subset of dict items that can be passed to function as kwargs

Parameters

- **config** (*Dict[str, Any]*) – a flat configuration dictionary
- **func** (*Callable*) – a function or method
- **start** (*int, default=0*) – Only take args after this position. Set to 1 if calling with an unbound method to avoid the `self` argument.

Returns a subset of `config` that only contains items compatible with the signature of `func`.

Return type Dict[str, Any]

Example

```
>>> # An example use case is to select a subset of of a config
>>> # that can be passed to some function as kwargs
>>> import ubelt as ub
>>> # Define a function with args that match some keys in a config.
>>> def func(a, e, f):
>>>     return a * e * f
>>> # Define a config that has a superset of items needed by the func
>>> config = {
...   'a': 2, 'b': 3, 'c': 7,
...   'd': 11, 'e': 13, 'f': 17,
... }
>>> # Call the function only with keys that are compatible
>>> func(**ub.compatible(config, func))
442
```

Example

```
>>> # Test case with kwargs
>>> import ubelt as ub
>>> def func(a, e, f, *args, **kwargs):
>>>     return a * e * f
>>> config = {
...   'a': 2, 'b': 3, 'c': 7,
...   'd': 11, 'e': 13, 'f': 17,
... }
>>> func(**ub.compatible(config, func))
```

1.27.1.1.1.13 ubelt.util_futures module

Introduces the Executor class that wraps the standard ThreadPoolExecutor, ProcessPoolExecutor, and the new SerialExecutor with a common interface and a backend that changes dynamically. This makes is easy to test if your code benefits from parallism, how much it benefits, and gives you the ability to disable if if you need to.

Example

```
>>> # xdoctest: +SKIP
>>> # Note: while this works in IPython, this does not work when running
>>> # in xdoctest. https://github.com/Erotemic/xdoctest/issues/101
>>> # xdoctest: +REQUIRES(module:timerit)
>>> # Does my function benefit from parallelism?
>>> def my_function(arg1, arg2):
...     return (arg1 + arg2) * 3
>>> #
>>> def run_process(inputs, mode='serial', max_workers=0):
...     from concurrent.futures import as_completed
...     import ubelt as ub
...     # The executor interface is the same regardless of modes
...     executor = ub.Executor(mode=mode, max_workers=max_workers)
...     # submit returns a Future object
...     jobs = [executor.submit(my_function, *args) for args in inputs]
...     # future objects will contain results when they are done
...     results = [job.result() for job in as_completed(jobs)]
...     return results
>>> # The same code tests our method in serial, thread, or process mode
>>> import timerit
>>> ti = timerit.Timerit(100, bestof=10, verbose=2)
>>> # Setup test data
>>> import random
>>> rng = random.Random(0)
>>> max_workers = 4
>>> inputs = [(rng.random(), rng.random()) for _ in range(100)]
>>> for mode in ['serial', 'process', 'thread']:
>>>     for timer in ti.reset('mode={} max_workers={}'.format(mode, max_workers)):
>>>         with timer:
>>>             run_process(inputs, mode=mode, max_workers=max_workers)
>>> print(ub.repr2(ti))
```

class ubelt.util_futures.**Executor**(*mode='thread', max_workers=0*)

Bases: object

Wrapper around a specific executor.

Abstracts Serial, Thread, and Process Executor via arguments.

Parameters

- **mode** (*str, default='thread'*) – either thread, serial, or process
- **max_workers** (*int, default=0*) – number of workers. If 0, serial is forced.

Example

```
>>> import platform
>>> import sys
>>> # The process backend breaks pyp3 when using coverage
>>> if 'pypy' in platform.python_implementation().lower():
...     import pytest
...     pytest.skip('not testing process on pypy')
>>> if sys.platform.startswith('win32'):
...     import pytest
...     pytest.skip('not running this test on win32 for now')
>>> import ubelt as ub
>>> # Fork before threading!
>>> # https://pybay.com/site_media/slides/raymond2017-keynote/combo.html
>>> self1 = ub.Executor(mode='serial', max_workers=0)
>>> self1.__enter__()
>>> self2 = ub.Executor(mode='process', max_workers=2)
>>> self2.__enter__()
>>> self3 = ub.Executor(mode='thread', max_workers=2)
>>> self3.__enter__()
>>> jobs = []
>>> jobs.append(self1.submit(sum, [1, 2, 3]))
>>> jobs.append(self1.submit(sum, [1, 2, 3]))
>>> jobs.append(self2.submit(sum, [10, 20, 30]))
>>> jobs.append(self2.submit(sum, [10, 20, 30]))
>>> jobs.append(self3.submit(sum, [4, 5, 5]))
>>> jobs.append(self3.submit(sum, [4, 5, 5]))
>>> for job in jobs:
>>>     result = job.result()
>>>     print('result = {!r}'.format(result))
>>> self1.__exit__(None, None, None)
>>> self2.__exit__(None, None, None)
>>> self3.__exit__(None, None, None)
```

Example

```
>>> import ubelt as ub
>>> self1 = ub.Executor(mode='serial', max_workers=0)
>>> with self1:
>>>     jobs = []
>>>     for i in range(10):
>>>         jobs.append(self1.submit(sum, [i + 1, i]))
>>>     for job in jobs:
>>>         job.add_done_callback(lambda x: print('done callback got x = {}'.
↪format(x)))
>>>         result = job.result()
>>>         print('result = {!r}'.format(result))
```

submit(*func*, **args*, ***kw*)

Calls the submit function of the underlying backend.

Returns a future representing the job

Return type concurrent.futures.Future

shutdown()

Calls the shutdown function of the underlying backend.

map(*fn*, **iterables*, ***kwargs*)

Calls the map function of the underlying backend.

CommandLine

```
xdoctest -m /home/joncrall/code/ubelt/ubelt/util_futures.py Executor.map
```

Example

```
>>> import ubelt as ub
>>> import concurrent.futures
>>> import string
>>> with ub.Executor(mode='serial') as executor:
...     result_iter = executor.map(int, string.digits)
...     results = list(result_iter)
>>> print('results = {!r}'.format(results))
results = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> with ub.Executor(mode='thread', max_workers=2) as executor:
...     result_iter = executor.map(int, string.digits)
...     results = list(result_iter)
>>> # xdoctest: +IGNORE_WANT
>>> print('results = {!r}'.format(results))
results = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

class ubelt.util_futures.**JobPool**(*mode='thread'*, *max_workers=0*)

Bases: object

Abstracts away boilerplate of submitting and collecting jobs

This is a basic wrapper around *ubelt.util_futures.Executor* that simplifies the most basic case.

Example

```
>>> import ubelt as ub
>>> def worker(data):
>>>     return data + 1
>>> pool = ub.JobPool('thread', max_workers=16)
>>> for data in ub.ProgIter(range(10), desc='submit jobs'):
>>>     pool.submit(worker, data)
>>> final = []
>>> for job in pool.as_completed(desc='collect jobs'):
>>>     info = job.result()
>>>     final.append(info)
>>> print('final = {!r}'.format(final))
```

submit(*func*, **args*, ***kwargs*)

Submit a job managed by the pool

Parameters

- **func** (*Callable[..., Any]*) – A callable that will take as many arguments as there are passed iterables.
- ***args** – positional arguments to pass to the function
- ***kwargs** – keyword arguments to pass to the function

Returns a future representing the job

Return type concurrent.futures.Future

shutdown()

as_completed(*timeout=None*, *desc=None*, *progkw=None*)

Generates completed jobs in an arbitrary order

Parameters

- **timeout** (*float | None*) – Specify the the maximum number of seconds to wait for a job.
- **desc** (*str | None*) – if specified, reports progress with a `ubelt.progiter.ProgIter` object.
- **progkw** (*dict | None*) – extra keyword arguments to `ubelt.progiter.ProgIter`.

Yields *concurrent.futures.Future* – The completed future object containing the results of a job.

CommandLine

```
xdoctest -m ubelt.util_futures JobPool.as_completed
```

Example

```
>>> import ubelt as ub
>>> pool = ub.JobPool('thread', max_workers=8)
>>> text = ub.paragraph(
...     '''
...     UDP is a cool protocol, check out the wiki:
...
...     UDP-based Data Transfer Protocol (UDT), is a high-performance
...     data transfer protocol designed for transferring large
...     volumetric datasets over high-speed wide area networks. Such
...     settings are typically disadvantageous for the more common TCP
...     protocol.
...     ''')
>>> for word in text.split(' '):
...     pool.submit(print, word)
>>> for _ in pool.as_completed():
...     pass
>>> pool.shutdown()
```

join(***kwargs*)

Like **:method:`JobPool.as_completed`**, but executes the *result* method of each future and returns only after all processes are complete. This allows for lower-boilerplate prototyping.

Parameters ****kwargs** – passed to **:method:`JobPool.as_completed`**

Returns list of results

Return type List[Any]

Example

```
>>> import ubelt as ub
>>> # We just want to try replacing our simple iterative algorithm
>>> # with the embarassingly parallel version
>>> arglist = list(zip(range(1000), range(1000)))
>>> func = ub.identity
>>> #
>>> # Original version
>>> for args in arglist:
>>>     func(*args)
>>> #
>>> # Potentially parallel version
>>> jobs = ub.JobPool(max_workers=0)
>>> for args in arglist:
>>>     jobs.submit(func, *args)
>>> _ = jobs.join(desc='running')
```

1.27.1.1.1.14 ubelt.util_hash module

Wrappers around hashlib functions to generate hash signatures for common data.

The hashes are deterministic across python versions and operating systems. This is verified by CI testing on Windows, Linux, Python with 2.7, 3.4, and greater, and on 32 and 64 bit versions.

Use Case #1: You have data that you want to hash. If we assume the data is in standard python scalars or ordered sequences: e.g. tuple, list, odict, oset, int, str, etc..., then the solution is *hash_data()*.

Use Case #2: You have a file you want to hash, but your system doesn't have a sha1sum executable (or you dont want to use Popen). The solution is *hash_file()*

The *ubelt.util_hash.hash_data()* function recursively hashes most builtin python data structures.

The *ubelt.util_hash.hash_file()* function hashes data on disk. Both of the aforementioned functions have options for different hashers and alphabets.

Example

```
>>> import ubelt as ub
>>> data = ub.odict(sorted({
>>>     'param1': True,
>>>     'param2': 0,
>>>     'param3': [None],
>>>     'param4': ('str', 4.2),
>>> }.items()))
>>> # hash_data can hash any ordered builtin object
>>> ub.hash_data(data, hasher='sha256')
0b101481e4b894ddf6de57...
```

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> fpath = ub.touch(join(ub.ensure_app_cache_dir('ubelt'), 'empty_file'))
>>> ub.hash_file(fpath, hasher='sha1')
da39a3ee5e6b4b0d3255bfef95601890afd80709
```

Note: The exact hashes generated for data object and files may change in the future. When this happens the `HASH_VERSION` attribute will be incremented.

Note: [util_hash.Note.1] pre 0.10.2, the protected function _hashable_sequence defaulted to types=True setting to True here for backwards compat. This means that extensions using the `_hashable_sequence` helper will always include types in their hashable encoding regardless of the argument setting. We may change this in the future, to be more consistent. This is a minor detail unless you are getting into the weeds of how we coerce technically non-hashable sequences into a hashable encoding.

`ubelt.util_hash.`**`hash_data`**(*data*, *hasher=NoParam*, *base=NoParam*, *types=False*, *convert=False*, *extensions=None*)

Get a unique hash depending on the state of the data.

Parameters

- **data** (*object*) – Any sort of loosely organized data
- **hasher** (*str | Hasher, default='sha512'*) – string code or a hash algorithm from hashlib. Valid hashing algorithms are defined by `hashlib.algorithms_guaranteed` (e.g. 'sha1', 'sha512', 'md5') as well as 'xxh32' and 'xxh64' if `xxhash` is installed.
- **base** (*List[str] | str, default='hex'*) – list of symbols or shorthand key. Valid keys are 'abc', 'hex', and 'dec'.
- **types** (*bool*) – If True data types are included in the hash, otherwise only the raw data is hashed. Defaults to False.
- **convert** (*bool, default=True*) – if True, try and convert the data to json an the json is hashed instead. This can improve runtime in some instances, however the hash may differ from the case where convert=False.
- **extensions** (*HashableExtensions*) – a custom `HashableExtensions` instance that can overwrite or define how different types of objects are hashed.

Note: The types allowed are specified by the HashableExtensions object. By default ubelt will register:

OrderedDict, uuid.UUID, np.random.RandomState, np.int64, np.int32, np.int16, np.int8, np.uint64, np.uint32, np.uint16, np.uint8, np.float16, np.float32, np.float64, np.float128, np.ndarray, bytes, str, int, float, long (in python2), list, tuple, set, and dict

---

Returns text representing the hashed data

Return type str

Note: The alphabet26 base is a pretty nice base, I recommend it. However we default to `base='hex'` because it is standard. You can try the alphabet26 base by setting `base='abc'`.

Example

```
>>> import ubelt as ub
>>> print(ub.hash_data([1, 2, (3, '4')], convert=False))
60b758587f599663931057e6ebdf185a...
>>> print(ub.hash_data([1, 2, (3, '4')], base='abc',  hasher='sha512')[:32])
hsrgqvfiuxvvhcdnypivhhthmrolkzej
```

`ubelt.util_hash.`**`hash_file`**(*fpath, blocksize=1048576, stride=1, maxbytes=None, hasher=NoParam, base=NoParam*)

Hashes the data in a file on disk.

The results of this function agree with the standard UNIX commands (e.g. sha1sum, sha512sum, md5sum, etc…)

Parameters

- **fpath** (*PathLike*) – location of the file to be hashed.

- **blocksize** (*int, default=2 * 20*) – Amount of data to read and hash at a time. There is a trade off and the optimal number will depend on specific hardware. This number was chosen to be optimal on a developer system. See "dev/bench_hash_file" for methodology to choose this number for your use case.
- **stride** (*int, default=1*) – strides > 1 skip data to hash, useful for faster hashing, but less accurate, also makes hash dependent on blocksize.
- **maxbytes** (*int | None*) – if specified, only hash the leading *maxbytes* of data in the file.
- **hasher** (*str | Hasher, default='sha512'*) – string code or a hash algorithm from hashlib. Valid hashing algorithms are defined by `hashlib.algorithms_guaranteed` (e.g. 'sha1', 'sha512', 'md5') as well as 'xxh32' and 'xxh64' if `xxhash` is installed.

  TODO: add logic such that you can update an existing hasher
- **base** (*List[str] | str, default='hex'*) – list of symbols or shorthand key. Valid keys are 'abc', 'hex', and 'dec'.

Note: For better hashes keep stride = 1. For faster hashes set stride > 1. Blocksize matters when stride > 1.

References

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> dpath = ub.Path.appdir('ubelt/tests/test-hash').ensuredir()
>>> fpath = dpath / 'tmp1.txt'
>>> fpath.write_text('foobar')
>>> print(ub.hash_file(fpath, hasher='sha1', base='hex'))
8843d7f92416211de9ebb963ff4ce28125932878
```

Example

```
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('ubelt/tests/test-hash').ensuredir()
>>> fpath = dpath / 'tmp2.txt'
>>> # We have the ability to only hash at most ``maxbytes`` in a file
>>> fpath.write_text('abcdefghijklmnop')
>>> h0 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=11, blocksize=3)
>>> h1 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=32, blocksize=3)
>>> h2 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=32, blocksize=32)
>>> h3 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=16, blocksize=1)
>>> h4 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=16, blocksize=18)
>>> assert h1 == h2 == h3 == h4
>>> assert h1 != h0
```

```
>>> # Using a stride makes the result dependent on the blocksize
>>> h0 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=11, blocksize=3,␣
↪stride=2)
```

(continues on next page)

(continued from previous page)

```
>>> h1 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=32, blocksize=3,␣
↪stride=2)
>>> h2 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=32, blocksize=32,␣
↪stride=2)
>>> h3 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=16, blocksize=1,␣
↪stride=2)
>>> h4 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=16, blocksize=18,␣
↪stride=2)
>>> assert h1 != h2 != h3
>>> assert h1 == h0
>>> assert h2 == h4
```

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> dpath = ub.ensure_app_cache_dir('ubelt/tests/test-hash')
>>> fpath = ub.touch(join(dpath, 'empty_file'))
>>> # Test that the output is the same as sha1sum executable
>>> if ub.find_exe('sha1sum'):
>>>     want = ub.cmd(['sha1sum', fpath], verbose=2)['out'].split(' ')[0]
>>>     got = ub.hash_file(fpath, hasher='sha1')
>>>     print('want = {!r}'.format(want))
>>>     print('got = {!r}'.format(got))
>>>     assert want.endswith(got)
>>> # Do the same for sha512 sum and md5sum
>>> if ub.find_exe('sha512sum'):
>>>     want = ub.cmd(['sha512sum', fpath], verbose=2)['out'].split(' ')[0]
>>>     got = ub.hash_file(fpath, hasher='sha512')
>>>     print('want = {!r}'.format(want))
>>>     print('got = {!r}'.format(got))
>>>     assert want.endswith(got)
>>> if ub.find_exe('md5sum'):
>>>     want = ub.cmd(['md5sum', fpath], verbose=2)['out'].split(' ')[0]
>>>     got = ub.hash_file(fpath, hasher='md5')
>>>     print('want = {!r}'.format(want))
>>>     print('got = {!r}'.format(got))
>>>     assert want.endswith(got)
```

1.27.1.1.1.15 ubelt.util_import module

Expose functions to simplify importing from module names and paths.

The *ubelt.import_module_from_path()* function does its best to load a python file into th current set of global modules.

The *ubelt.import_module_from_name()* works similarly.

The *ubelt.modname_to_modpath()* and *ubelt.modname_to_modpath()* work statically and convert between module names and file paths on disk.

The *ubelt.split_modpath()* function separates modules into a root and base path depending on where the first `__init__.py` file is.

`ubelt.util_import.`**`split_modpath`**(*modpath*, *check=True*)

Splits the modpath into the dir that must be in PYTHONPATH for the module to be imported and the modulepath relative to this directory.

Parameters

- **modpath** (*str*) – module filepath
- **check** (*bool*) – if False, does not raise an error if modpath is a directory and does not contain an `__init__.py` file.

Returns (directory, rel_modpath)

Return type Tuple[str, str]

Raises `ValueError` – if modpath does not exist or is not a package

Example

```
>>> from xdoctest import static_analysis
>>> modpath = static_analysis.__file__.replace('.pyc', '.py')
>>> modpath = abspath(modpath)
>>> dpath, rel_modpath = split_modpath(modpath)
>>> recon = join(dpath, rel_modpath)
>>> assert recon == modpath
>>> assert rel_modpath == join('xdoctest', 'static_analysis.py')
```

`ubelt.util_import.`**`modname_to_modpath`**(*modname*, *hide_init=True*, *hide_main=False*, *sys_path=None*)

Finds the path to a python module from its name.

Determines the path to a python module without directly import it

Converts the name of a module (__name__) to the path (__file__) where it is located without importing the module. Returns None if the module does not exist.

Parameters

- **modname** (*str*) – module filepath
- **hide_init** (*bool*) – if False, __init__.py will be returned for packages
- **hide_main** (*bool*) – if False, and hide_init is True, __main__.py will be returned for packages, if it exists.
- **sys_path** (*list, default=None*) – if specified overrides `sys.path`

Returns modpath - path to the module, or None if it doesn't exist

Return type str

Example

```
>>> modname = 'xdoctest.__main__'
>>> modpath = modname_to_modpath(modname, hide_main=False)
>>> assert modpath.endswith('__main__.py')
>>> modname = 'xdoctest'
>>> modpath = modname_to_modpath(modname, hide_init=False)
>>> assert modpath.endswith('__init__.py')
>>> # xdoctest: +REQUIRES(CPython)
>>> modpath = basename(modname_to_modpath('_ctypes'))
>>> assert 'ctypes' in modpath
```

ubelt.util_import.**modpath_to_modname**(*modpath*, *hide_init=True*, *hide_main=False*, *check=True*, *relativeto=None*)

Determines importable name from file path

Converts the path to a module (__file__) to the importable python name (__name__) without importing the module.

The filename is converted to a module name, and parent directories are recursively included until a directory without an __init__.py file is encountered.

Parameters

- **modpath** (*str*) – module filepath
- **hide_init** (*bool, default=True*) – removes the __init__ suffix
- **hide_main** (*bool, default=False*) – removes the __main__ suffix
- **check** (*bool, default=True*) – if False, does not raise an error if modpath is a dir and does not contain an __init__ file.
- **relativeto** (*str, default=None*) – if specified, all checks are ignored and this is considered the path to the root module.

Todo:

- **[ ] Does this need modification to support PEP 420?** https://www.python.org/dev/peps/pep-0420/

Returns modname

Return type str

Raises `ValueError` – if check is True and the path does not exist

Example

```
>>> from xdoctest import static_analysis
>>> modpath = static_analysis.__file__.replace('.pyc', '.py')
>>> modpath = modpath.replace('.pyc', '.py')
>>> modname = modpath_to_modname(modpath)
>>> assert modname == 'xdoctest.static_analysis'
```

Example

```
>>> import xdoctest
>>> assert modpath_to_modname(xdoctest.__file__.replace('.pyc', '.py')) == 'xdoctest
↪'
>>> assert modpath_to_modname(dirname(xdoctest.__file__.replace('.pyc', '.py'))) ==
↪'xdoctest'
```

Example

```
>>> # xdoctest: +REQUIRES(CPython)
>>> modpath = modname_to_modpath('_ctypes')
>>> modname = modpath_to_modname(modpath)
>>> assert modname == '_ctypes'
```

Example

```
>>> modpath = '/foo/libfoobar.linux-x86_64-3.6.so'
>>> modname = modpath_to_modname(modpath, check=False)
>>> assert modname == 'libfoobar'
```

ubelt.util_import.**import_module_from_name**(*modname*)

Imports a module from its string name (i.e. `__name__`)

Parameters **modname** (*str*) – module name

Returns module

Return type ModuleType

SeeAlso: *import_module_from_path()*

Example

```
>>> # test with modules that won't be imported in normal circumstances
>>> # todo write a test where we guarantee this
>>> modname_list = [
>>>     'pickletools',
>>>     'lib2to3.fixes.fix_apply',
>>> ]
>>> #assert not any(m in sys.modules for m in modname_list)
>>> modules = [import_module_from_name(modname) for modname in modname_list]
>>> assert [m.__name__ for m in modules] == modname_list
>>> assert all(m in sys.modules for m in modname_list)
```

ubelt.util_import.**import_module_from_path**(*modpath*, *index=- 1*)

Imports a module via its path

Parameters

- **modpath** (*str | PathLike*) – Path to the module on disk or within a zipfile.

- **index** (*int*) – Location at which we modify PYTHONPATH if necessary. If your module name does not conflict, the safest value is -1, However, if there is a conflict, then use an index of 0. The default may change to 0 in the future.

Returns the imported module

Return type ModuleType

References

Note: If the module is part of a package, the package will be imported first. These modules may cause problems when reloading via IPython magic

This can import a module from within a zipfile. To do this modpath should specify the path to the zipfile and the path to the module within that zipfile separated by a colon or pathsep. E.g. "/path/to/archive.zip:mymodule.pl"

Warning: It is best to use this with paths that will not conflict with previously existing modules.

If the modpath conflicts with a previously existing module name. And the target module does imports of its own relative to this conflicting path. In this case, the module that was loaded first will win.

For example if you try to import '/foo/bar/pkg/mod.py' from the folder structure:

```
- foo/
  +- bar/
     +- pkg/
        +  __init__.py
        |- mod.py
        |- helper.py
```

If there exists another module named `pkg` already in sys.modules and mod.py does something like `from . import helper`, Python will assume helper belongs to the `pkg` module already in sys.modules. This can cause a NameError or worse — a incorrect helper module.

SeeAlso: *import_module_from_name()*

Example

```
>>> import xdoctest
>>> modpath = xdoctest.__file__
>>> module = import_module_from_path(modpath)
>>> assert module is xdoctest
```

Example

```
>>> # Test importing a module from within a zipfile
>>> import zipfile
>>> from xdoctest import utils
>>> from os.path import join, expanduser, normpath
>>> dpath = expanduser('~/.cache/xdoctest')
>>> dpath = utils.ensuredir(dpath)
>>> #dpath = utils.TempDir().ensure()
>>> # Write to an external module named bar
>>> external_modpath = join(dpath, 'bar.py')
>>> # For pypy support we have to write this using with
>>> with open(external_modpath, 'w') as file:
>>>     file.write('testvar = 1')
>>> internal = 'folder/bar.py'
>>> # Move the external bar module into a zipfile
>>> zippath = join(dpath, 'myzip.zip')
>>> with zipfile.ZipFile(zippath, 'w') as myzip:
>>>     myzip.write(external_modpath, internal)
>>> # Import the bar module from within the zipfile
>>> modpath = zippath + ':' + internal
>>> modpath = zippath + os.path.sep + internal
>>> module = import_module_from_path(modpath)
>>> assert normpath(module.__name__) == normpath('folder/bar')
>>> assert module.testvar == 1
```

Example

```
>>> import pytest
>>> with pytest.raises(IOError):
>>>     import_module_from_path('does-not-exist')
>>> with pytest.raises(IOError):
>>>     import_module_from_path('does-not-exist.zip/')
```

1.27.1.1.1.16 ubelt.util_indexable module

The util_indexable module defines `IndexableWalker` which is a powerful way to iterate through nested Python containers.

class ubelt.util_indexable.**IndexableWalker**(*data*, *dict_cls=(<class 'dict'>,)*, *list_cls=(<class 'list'>, <class 'tuple'>)*)

Bases: `collections.abc.Generator`

Traverses through a nested tree-liked indexable structure.

Generates a path and value to each node in the structure. The path is a list of indexes which if applied in order will reach the value.

The `__setitem__` method can be used to modify a nested value based on the path returned by the generator.

When generating values, you can use "send" to prevent traversal of a particular branch.

RelatedWork:

- **https://pypi.org/project/python-benedict/ - implements a dictionary** subclass with similar nested indexing abilities.

Example

```
>>> # Given Nested Data
>>> data = {
>>>     'foo': {'bar': 1},
>>>     'baz': [{'biz': 3}, {'buz': [4, 5, 6]}],
>>> }
>>> # Create an IndexableWalker
>>> walker = IndexableWalker(data)
>>> # We iterate over the data as if it was flat
>>> # ignore the <want> string due to order issues on older Pythons
>>> # xdoctest: +IGNORE_WANT
>>> for path, val in walker:
>>>     print(path)
['foo']
['baz']
['baz', 0]
['baz', 1]
['baz', 1, 'buz']
['baz', 1, 'buz', 0]
['baz', 1, 'buz', 1]
['baz', 1, 'buz', 2]
['baz', 0, 'biz']
['foo', 'bar']
>>> # We can use "paths" as keys to getitem into the walker
>>> path = ['baz', 1, 'buz', 2]
>>> val = walker[path]
>>> assert val == 6
>>> # We can use "paths" as keys to setitem into the walker
>>> assert data['baz'][1]['buz'][2] == 6
>>> walker[path] = 7
>>> assert data['baz'][1]['buz'][2] == 7
>>> # We can use "paths" as keys to delitem into the walker
>>> assert data['baz'][1]['buz'][1] == 5
>>> del walker[['baz', 1, 'buz', 1]]
>>> assert data['baz'][1]['buz'][1] == 7
```

Example

```
>>> # Create nested data
>>> # xdoctest: +REQUIRES(module:numpy)
>>> import numpy as np
>>> import ubelt as ub
>>> data = ub.ddict(lambda: int)
>>> data['foo'] = ub.ddict(lambda: int)
>>> data['bar'] = np.array([1, 2, 3])
>>> data['foo']['a'] = 1
>>> data['foo']['b'] = np.array([1, 2, 3])
```

(continues on next page)

(continued from previous page)

```
>>> data['foo']['c'] = [1, 2, 3]
>>> data['baz'] = 3
>>> print('data = {}'.format(ub.repr2(data, nl=True)))
>>> # We can walk through every node in the nested tree
>>> walker = IndexableWalker(data)
>>> for path, value in walker:
>>>     print('walk path = {}'.format(ub.repr2(path, nl=0)))
>>>     if path[-1] == 'c':
>>>         # Use send to prevent traversing this branch
>>>         got = walker.send(False)
>>>         # We can modify the value based on the returned path
>>>         walker[path] = 'changed the value of c'
>>> print('data = {}'.format(ub.repr2(data, nl=True)))
>>> assert data['foo']['c'] == 'changed the value of c'
```

Example

```
>>> # Test sending false for every data item
>>> # xdoctest: +REQUIRES(CPython)
>>> # xdoctest: +REQUIRES(module:numpy)
>>> import ubelt as ub
>>> import numpy as np
>>> data = {1: 1}
>>> walker = IndexableWalker(data)
>>> for path, value in walker:
>>>     print('walk path = {}'.format(ub.repr2(path, nl=0)))
>>>     walker.send(False)
>>> data = {}
>>> walker = IndexableWalker(data)
>>> for path, value in walker:
>>>     walker.send(False)
```

send(*arg*) → send 'arg' into generator,

return next yielded value or raise StopIteration.

throw(*typ*[, *val*[, *tb*]]) → raise exception in generator,

return next yielded value or raise StopIteration.

ubelt.util_indexable.**indexable_allclose**(*dct1*, *dct2*, *rel_tol=1e-09*, *abs_tol=0.0*, *return_info=False*)

Walks through two nested data structures and ensures that everything is roughly the same.

Parameters

- **dct1** (*dict*) – a nested indexable item
- **dct2** (*dict*) – a nested indexable item
- **rel_tol** (*float*) – maximum difference for being considered "close", relative to the magnitude of the input values
- **abs_tol** (*float*) – maximum difference for being considered "close", regardless of the magnitude of the input values
- **return_info** (*bool, default=False*) – if true, return extra info

Returns A boolean result if `return_info` is false, otherwise a tuple of the boolean result and an "info" dict containing detailed results indicating what matched and what did not.

Return type bool | Tuple[bool, Dict]

Example

```
>>> import ubelt as ub
>>> dct1 = {
>>>     'foo': [1.222222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> dct2 = {
>>>     'foo': [1.22222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> flag, return_info =  ub.indexable_allclose(dct1, dct2, return_info=True)
>>> print('return_info = {}'.format(ub.repr2(return_info, nl=1)))
>>> print('flag = {!r}'.format(flag))
```

```
>>> walker1 = return_info['walker1']
>>> for p1, v1, v2  in return_info['faillist']:
>>>     v1_ = walker1[p1]
>>>     print('*fail p1, v1, v2 = {}, {}, {}'.format(p1, v1, v2))
>>> for p1 in return_info['passlist']:
>>>     v1_ = walker1[p1]
>>>     print('*pass p1, v1_ = {}, {}'.format(p1, v1_))
>>> assert not flag
```

```
>>> import ubelt as ub
>>> dct1 = {
>>>     'foo': [1.000000000000000000000001, 1.],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> dct2 = {
>>>     'foo': [0.9999999999999999, 1.],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> flag, return_info =  ub.indexable_allclose(dct1, dct2, return_info=True)
>>> print('return_info = {}'.format(ub.repr2(return_info, nl=1)))
>>> print('flag = {!r}'.format(flag))
```

Example

```
>>> import ubelt as ub
>>> flag, return_info =  ub.indexable_allclose([], [], return_info=True)
>>> print('return_info = {!r}'.format(return_info))
>>> print('flag = {!r}'.format(flag))
```

Example

```
>>> import ubelt as ub
>>> flag =  ub.indexable_allclose([], [], return_info=False)
>>> print('flag = {!r}'.format(flag))
```

Example

```
>>> import ubelt as ub
>>> flag, return_info =  ub.indexable_allclose([], [1], return_info=True)
>>> print('return_info = {!r}'.format(return_info))
>>> print('flag = {!r}'.format(flag))
```

1.27.1.1.1.17 ubelt.util_io module

Functions for reading and writing files on disk.

writeto() and *readfrom()* wrap `open().write()` and `open().read()` and primarily serve to indicate that the type of data being written and read is unicode text.

delete() wraps `os.unlink()` and `shutil.rmtree()` and does not throw an error if the file or directory does not exist. It also contains workarounds for win32 issues with `shutil`.

`ubelt.util_io.readfrom`(*fpath*, *aslines=False*, *errors='replace'*, *verbose=None*)

Reads (utf8) text from a file.

Note: You probably should use `ub.Path(<fpath>).read_text()` instead. This function exists as a convenience for writing in Python2. After 2020-01-01, we may consider deprecating the function.

Parameters

- **fpath** (*str | PathLike*) – file path
- **aslines** (*bool*) – if True returns list of lines
- **verbose** (*bool*) – verbosity flag

Returns text from fpath (this is unicode)

Return type str

`ubelt.util_io.writeto`(*fpath*, *to_write*, *aslines=False*, *verbose=None*)

Writes (utf8) text to a file.

Parameters

- **fpath** (*str | PathLike*) – file path
- **to_write** (*str*) – text to write (must be unicode text)
- **aslines** (*bool*) – if True to_write is assumed to be a list of lines
- **verbose** (*bool*) – verbosity flag

Note: In CPython you may want to use `open(<fpath>).write(<to_write>)` instead. This function exists as a convenience for writing in Python2. After 2020-01-01, we may consider deprecating the function.

NOTE: In PyPy `open(<fpath>).write(<to_write>)` does not work. See *https://pypy.org/compat.html*. This is an argument for keeping this function.

NOTE: With modern versions of Python, it is generally recommened to use `pathlib.Path.write_text()` instead. Although there does seem to be some corner case this handles better on win32, so maybe useful?

Example

```
>>> import ubelt as ub
>>> import os
>>> from os.path import exists
>>> dpath = ub.ensure_app_cache_dir('ubelt')
>>> fpath = dpath + '/' + 'testwrite.txt'
>>> if exists(fpath):
>>>     os.remove(fpath)
>>> to_write = 'utf-8 symbols , , , , , , , , and .'
>>> ub.writeto(fpath, to_write)
>>> read_ = ub.readfrom(fpath)
>>> print('read_    = ' + read_)
>>> print('to_write = ' + to_write)
>>> assert read_ == to_write
```

Example

```
>>> import ubelt as ub
>>> import os
>>> from os.path import exists
>>> dpath = ub.ensure_app_cache_dir('ubelt')
>>> fpath = dpath + '/' + 'testwrite2.txt'
>>> if exists(fpath):
>>>     os.remove(fpath)
>>> to_write = ['a\n', 'b\n', 'c\n', 'd\n']
>>> ub.writeto(fpath, to_write, aslines=True)
>>> read_ = ub.readfrom(fpath, aslines=True)
>>> print('read_    = {}'.format(read_))
>>> print('to_write = {}'.format(to_write))
>>> assert read_ == to_write
```

Example

```
>>> # With modern Python, use pathlib.Path (or ub.Path) instead
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('ubelt/tests/io').ensuredir()
>>> fpath = (dpath / 'test_file.txt').delete()
>>> to_write = 'utf-8 symbols , , , , , , , , and .'
>>> ub.writeto(fpath, to_write)
>>> fpath.write_bytes(to_write.encode('utf8'))
>>> assert fpath.read_bytes().decode('utf8') == to_write
```

`ubelt.util_io.touch`(*fpath*, *mode=438*, *dir_fd=None*, *verbose=0*, ***kwargs*)

change file timestamps

Works like the touch unix utility

Parameters

- **fpath** (*str | PathLike*) – name of the file
- **mode** (*int*) – file permissions (python3 and unix only)
- **dir_fd** (*io.IOBase*) – optional directory file descriptor. If specified, fpath is interpreted as relative to this descriptor (python 3 only).
- **verbose** (*int*) – verbosity
- ****kwargs** – extra args passed to `os.utime()` (python 3 only).

Returns path to the file

Return type str

References

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> dpath = ub.ensure_app_cache_dir('ubelt')
>>> fpath = join(dpath, 'touch_file')
>>> assert not exists(fpath)
>>> ub.touch(fpath)
>>> assert exists(fpath)
>>> os.unlink(fpath)
```

`ubelt.util_io.delete`(*path*, *verbose=False*)

Removes a file or recursively removes a directory. If a path does not exist, then this is does nothing.

Parameters

- **path** (*str | PathLike*) – file or directory to remove
- **verbose** (*bool*) – if True prints what is being done

SeeAlso:

send2trash - A cross-platform Python package for sending files to the trash instead of irreversibly deleting them.

ubelt.util_path.Path.delete()

Notes

This can call os.unlink(), os.rmdir(), or shutil.rmtree(), depending on what path references on the filesystem. (On windows may also call a custom ubelt._win32_links._win32_rmtree()).

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> base = ub.ensure_app_cache_dir('ubelt', 'delete_test')
>>> dpath1 = ub.ensuredir(join(base, 'dir'))
>>> ub.ensuredir(join(base, 'dir', 'subdir'))
>>> ub.touch(join(base, 'dir', 'to_remove1.txt'))
>>> fpath1 = join(base, 'dir', 'subdir', 'to_remove3.txt')
>>> fpath2 = join(base, 'dir', 'subdir', 'to_remove2.txt')
>>> ub.touch(fpath1)
>>> ub.touch(fpath2)
>>> assert all(map(exists, (dpath1, fpath1, fpath2)))
>>> ub.delete(fpath1)
>>> assert all(map(exists, (dpath1, fpath2)))
>>> assert not exists(fpath1)
>>> ub.delete(dpath1)
>>> assert not any(map(exists, (dpath1, fpath1, fpath2)))
```

Example

```
>>> import ubelt as ub
>>> from os.path import exists, join
>>> dpath = ub.ensure_app_cache_dir('ubelt', 'delete_test2')
>>> dpath1 = ub.ensuredir(join(dpath, 'dir'))
>>> fpath1 = ub.touch(join(dpath1, 'to_remove.txt'))
>>> assert exists(fpath1)
>>> ub.delete(dpath)
>>> assert not exists(fpath1)
```

1.27.1.1.1.18 ubelt.util_links module

Cross-platform logic for dealing with symlinks. Basic functionality should work on all operating systems including everyone's favorite pathological OS (note that there is an additional helper file for this case), but there are some corner cases depending on your version. Recent versions of Windows tend to work, but there certain system settings that cause issues. Any POSIX system works without difficulty.

Example

```
>>> import ubelt as ub
>>> from os.path import normpath, join
>>> dpath = ub.ensure_app_cache_dir('ubelt', normpath('demo/symlink'))
>>> real_path = join(dpath, 'real_file.txt')
>>> link_path = join(dpath, 'link_file.txt')
>>> ub.touch(real_path)
>>> result = ub.symlink(real_path, link_path, overwrite=True, verbose=3)
>>> parts = result.split(os.path.sep)
>>> print(parts[-1])
link_file.txt
```

`ubelt.util_links.`**`symlink`**(*real_path*, *link_path*, *overwrite=False*, *verbose=0*)

Create a link `link_path` that mirrors `real_path`.

This function attempts to create a real symlink, but will fall back on a hard link or junction if symlinks are not supported.

Parameters

- **path** (*str | PathLike*) – path to real file or directory
- **link_path** (*str | PathLike*) – path to desired location for symlink
- **overwrite** (*bool, default=False*) – overwrite existing symlinks. This will not overwrite real files on systems with proper symlinks. However, on older versions of windows junctions are indistinguishable from real files, so we cannot make this guarantee.
- **verbose** (*int, default=0*) – verbosity level

Returns link path

Return type str | PathLike

Note: On systems that do not contain support for symlinks (e.g. some versions / configurations of Windows), this function will fall back on hard links or junctions [WikiNTFSLinks], [WikiHardLink]. The differences between the two are explained in [WikiSymLink].

If symlinks are not available, then `link_path` and `real_path` must exist on the same filesystem. Given that, this function always works in the sense that (1) `link_path` will mirror the data from `real_path`, (2) updates to one will effect the other, and (3) no extra space will be used.

More details can be found in `ubelt._win32_links`. On systems that support symlinks (e.g. Linux), none of the above applies.

References

Example

```
>>> import ubelt as ub
>>> dpath = ub.ensure_app_cache_dir('ubelt', 'test_symlink0')
>>> real_path = join(dpath, 'real_file.txt')
>>> link_path = join(dpath, 'link_file.txt')
>>> [ub.delete(p) for p in [real_path, link_path]]
```

(continues on next page)

(continued from previous page)

```
>>> ub.writeto(real_path, 'foo')
>>> result = symlink(real_path, link_path)
>>> assert ub.readfrom(result) == 'foo'
>>> [ub.delete(p) for p in [real_path, link_path]]
```

Example

```
>>> import ubelt as ub
>>> from os.path import dirname
>>> dpath = ub.ensure_app_cache_dir('ubelt', 'test_symlink1')
>>> ub.delete(dpath)
>>> ub.ensuredir(dpath)
>>> _dirstats(dpath)
>>> real_dpath = ub.ensuredir((dpath, 'real_dpath'))
>>> link_dpath = ub.augpath(real_dpath, base='link_dpath')
>>> real_path = join(dpath, 'afile.txt')
>>> link_path = join(dpath, 'afile.txt')
>>> [ub.delete(p) for p in [real_path, link_path]]
>>> ub.writeto(real_path, 'foo')
>>> result = symlink(real_dpath, link_dpath)
>>> assert ub.readfrom(link_path) == 'foo', 'read should be same'
>>> ub.writeto(link_path, 'bar')
>>> _dirstats(dpath)
>>> assert ub.readfrom(link_path) == 'bar', 'very bad bar'
>>> assert ub.readfrom(real_path) == 'bar', 'changing link did not change real'
>>> ub.writeto(real_path, 'baz')
>>> _dirstats(dpath)
>>> assert ub.readfrom(real_path) == 'baz', 'very bad baz'
>>> assert ub.readfrom(link_path) == 'baz', 'changing real did not change link'
>>> ub.delete(link_dpath, verbose=1)
>>> _dirstats(dpath)
>>> assert not exists(link_dpath), 'link should not exist'
>>> assert exists(real_path), 'real path should exist'
>>> _dirstats(dpath)
>>> ub.delete(dpath, verbose=1)
>>> _dirstats(dpath)
>>> assert not exists(real_path)
```

1.27.1.1.1.19 ubelt.util_list module

Utility functions for manipulating iterables, lists, and sequences.

The *chunks()* function splits a list into smaller parts. There are different strategies for how to do this.

The *flatten()* function take a list of lists and removees the inner lists. This only removes one level of nesting.

The *iterable()* function checks if an object is iterable or not. Similar to the `callable()` builtin function.

The *argmax()*, *argmin()*, and *argsort()* work similarly to the analogous `numpy` functions, except they operate on dictionaries and other Python builtin types.

The *take()* and *compress()* are generators, and also similar to their lesser known, but very useful numpy equivalents.

There are also other numpy inspired functions: `unique()`, `argunique()`, `unique_flags()`, and `boolmask()`.

`ubelt.util_list.allsame(iterable, eq=<built-in function eq>)`

Determine if all items in a sequence are the same

Parameters

- **iterable** (*Iterable[T]*) – items to determine if they are all the same
- **eq** (*Callable[[T, T], bool], default=operator.eq*) – function used to test for equality

Returns True if all items are equal, otherwise False

Return type bool

Notes

Similar to `more_itertools.all_equal()`

Example

```
>>> import ubelt as ub
>>> ub.allsame([1, 1, 1, 1])
True
>>> ub.allsame([])
True
>>> ub.allsame([0, 1])
False
>>> iterable = iter([0, 1, 1, 1])
>>> next(iterable)
>>> ub.allsame(iterable)
True
>>> ub.allsame(range(10))
False
>>> ub.allsame(range(10), lambda a, b: True)
True
```

`ubelt.util_list.argmax(indexable, key=None)`

Returns index / key of the item with the largest value.

This is similar to `numpy.argmax()`, but it is written in pure python and works on both lists and dictionaries.

Parameters

- **indexable** (*Iterable[VT] | Mapping[KT, VT]*) – indexable to sort by
- **key** (*Callable[[VT], Any], default=None*) – customizes the ordering of the indexable

Returns the index of the item with the maximum value.

Return type int | KT

Example

```
>>> import ubelt as ub
>>> assert ub.argmax({'a': 3, 'b': 2, 'c': 100}) == 'c'
>>> assert ub.argmax(['a', 'c', 'b', 'z', 'f']) == 3
>>> assert ub.argmax([[0, 1], [2, 3, 4], [5]], key=len) == 1
>>> assert ub.argmax({'a': 3, 'b': 2, 3: 100, 4: 4}) == 3
>>> assert ub.argmax(iter(['a', 'c', 'b', 'z', 'f'])) == 3
```

`ubelt.util_list.argmin(indexable, key=None)`

Returns index / key of the item with the smallest value.

This is similar to `numpy.argmin()`, but it is written in pure python and works on both lists and dictionaries.

Parameters

- **indexable** (*Iterable[VT] | Mapping[KT, VT]*) – indexable to sort by
- **key** (*Callable[[VT], VT], default=None*) – customizes the ordering of the indexable

Returns the index of the item with the minimum value.

Return type int | KT

Example

```
>>> import ubelt as ub
>>> assert ub.argmin({'a': 3, 'b': 2, 'c': 100}) == 'b'
>>> assert ub.argmin(['a', 'c', 'b', 'z', 'f']) == 0
>>> assert ub.argmin([[0, 1], [2, 3, 4], [5]], key=len) == 2
>>> assert ub.argmin({'a': 3, 'b': 2, 3: 100, 4: 4}) == 'b'
>>> assert ub.argmin(iter(['a', 'c', 'A', 'z', 'f'])) == 2
```

`ubelt.util_list.argsort(indexable, key=None, reverse=False)`

Returns the indices that would sort a indexable object.

This is similar to `numpy.argsort()`, but it is written in pure python and works on both lists and dictionaries.

Parameters

- **indexable** (*Iterable[VT] | Mapping[KT, VT]*) – indexable to sort by
- **key** (*Callable[[VT], VT] | None, default=None*) – customizes the ordering of the indexable
- **reverse** (*bool, default=False*) – if True returns in descending order

Returns indices - list of indices that sorts the indexable

Return type List[int] | List[KT]

Example

```
>>> import ubelt as ub
>>> # argsort works on dicts by returning keys
>>> dict_ = {'a': 3, 'b': 2, 'c': 100}
>>> indices = ub.argsort(dict_)
>>> assert list(ub.take(dict_, indices)) == sorted(dict_.values())
>>> # argsort works on lists by returning indices
>>> indexable = [100, 2, 432, 10]
>>> indices = ub.argsort(indexable)
>>> assert list(ub.take(indexable, indices)) == sorted(indexable)
>>> # Can use iterators, but be careful. It exhausts them.
>>> indexable = reversed(range(100))
>>> indices = ub.argsort(indexable)
>>> assert indices[0] == 99
>>> # Can use key just like sorted
>>> indexable = [[0, 1, 2], [3, 4], [5]]
>>> indices = ub.argsort(indexable, key=len)
>>> assert indices == [2, 1, 0]
>>> # Can use reverse just like sorted
>>> indexable = [0, 2, 1]
>>> indices = ub.argsort(indexable, reverse=True)
>>> assert indices == [1, 2, 0]
```

`ubelt.util_list.argunique`(*items*, *key=None*)

Returns indices corresponding to the first instance of each unique item.

Parameters

- **items** (*Sequence[VT]*) – indexable collection of items
- **key** (*Callable[[VT], Any], default=None*) – custom normalization function. If specified returns items where `key(item)` is unique.

Returns indices of the unique items

Return type Iterator[int]

Example

```
>>> import ubelt as ub
>>> items = [0, 2, 5, 1, 1, 0, 2, 4]
>>> indices = list(ub.argunique(items))
>>> assert indices == [0, 1, 2, 3, 7]
>>> indices = list(ub.argunique(items, key=lambda x: x % 2 == 0))
>>> assert indices == [0, 2]
```

`ubelt.util_list.boolmask`(*indices*, *maxval=None*)

Constructs a list of booleans where an item is True if its position is in `indices` otherwise it is False.

Parameters

- **indices** (*List[int]*) – list of integer indices
- **maxval** (*int*) – length of the returned list. If not specified this is inferred using `max(indices)`

Returns mask - a list of booleans. mask[idx] is True if idx in indices

Return type List[bool]

Note: In the future the arg `maxval` may change its name to `shape`

Example

```
>>> import ubelt as ub
>>> indices = [0, 1, 4]
>>> mask = ub.boolmask(indices, maxval=6)
>>> assert mask == [True, True, False, False, True, False]
>>> mask = ub.boolmask(indices)
>>> assert mask == [True, True, False, False, True]
```

class ubelt.util_list.**chunks**(*items*, *chunksize=None*, *nchunks=None*, *total=None*, *bordermode='none'*, *legacy=False*)

Bases: `object`

Generates successive n-sized chunks from `items`.

If the last chunk has less than n elements, `bordermode` is used to determine fill values.

Parameters

- **items** (*Iterable[T]*) – input to iterate over
- **chunksize** (*int*) – size of each sublist yielded
- **nchunks** (*int*) – number of chunks to create (cannot be specified if chunksize is specified)
- **bordermode** (*str*) – determines how to handle the last case if the length of the input is not divisible by chunksize valid values are: {'none', 'cycle', 'replicate'}
- **total** (*int*) – hints about the length of the input

Note:

FIXME: When nchunks is given, that's how many chunks we should get but the issue is that chunksize is not well defined in that instance For instance how do we turn a list with 4 elements into 3 chunks where does the extra item go?

In ubelt <= 0.10.3 there is a bug when specifying nchunks, where it chooses a chunksize that is too large. Specify `legacy=True` to get the old buggy behavior if needed.

Notes

This is similar to functionality provided by `more_itertools.chunked()`, `more_itertools.chunked_even()`, `more_itertools.sliced()`, `more_itertools.divide()`,

Yields *List[T]* – subsequent non-overlapping chunks of the input items

References

Example

```
>>> import ubelt as ub
>>> items = '1234567'
>>> genresult = ub.chunks(items, chunksize=3)
>>> list(genresult)
[['1', '2', '3'], ['4', '5', '6'], ['7']]
```

Example

```
>>> import ubelt as ub
>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> genresult = ub.chunks(items, chunksize=3, bordermode='none')
>>> assert list(genresult) == [[1, 2, 3], [4, 5, 6], [7]]
>>> genresult = ub.chunks(items, chunksize=3, bordermode='cycle')
>>> assert list(genresult) == [[1, 2, 3], [4, 5, 6], [7, 1, 2]]
>>> genresult = ub.chunks(items, chunksize=3, bordermode='replicate')
>>> assert list(genresult) == [[1, 2, 3], [4, 5, 6], [7, 7, 7]]
```

Example

```
>>> import ubelt as ub
>>> assert len(list(ub.chunks(range(2), nchunks=2))) == 2
>>> assert len(list(ub.chunks(range(3), nchunks=2))) == 2
>>> # Note: ub.chunks will not do the 2,1,1 split
>>> assert len(list(ub.chunks(range(4), nchunks=3))) == 3
>>> assert len(list(ub.chunks([], 2, bordermode='none'))) == 0
>>> assert len(list(ub.chunks([], 2, bordermode='cycle'))) == 0
>>> assert len(list(ub.chunks([], 2, None, bordermode='replicate'))) == 0
```

Example

```
>>> from ubelt.util_list import *  # NOQA
>>> def _check_len(self):
...     assert len(self) == len(list(self))
>>> _check_len(chunks(list(range(3)), nchunks=2))
>>> _check_len(chunks(list(range(2)), nchunks=2))
>>> _check_len(chunks(list(range(2)), nchunks=3))
```

Example

```
>>> from ubelt.util_list import *  # NOQA
>>> import pytest
>>> assert pytest.raises(ValueError, chunks, range(9))
>>> assert pytest.raises(ValueError, chunks, range(9), chunksize=2, nchunks=2)
>>> assert pytest.raises(TypeError, len, chunks((_ for _ in range(2)), 2))
```

Example

```
>>> from ubelt.util_list import *  # NOQA
>>> import ubelt as ub
>>> basis = {
>>>     'legacy': [False, True],
>>>     'chunker': [{'nchunks': 3}, {'nchunks': 4}, {'nchunks': 5}, {'nchunks': 7},
↪{'chunksize': 3}],
>>>     'items': [range(2), range(4), range(5), range(7), range(9)],
>>>     'bordermode': ['none', 'cycle', 'replicate'],
>>> }
>>> grid_items = list(ub.named_product(basis))
>>> rows = []
>>> for grid_item in ub.ProgIter(grid_items):
>>>     chunker = grid_item.get('chunker')
>>>     grid_item.update(chunker)
>>>     kw = ub.dict_diff(grid_item, {'chunker'})
>>>     self = chunk_iter = ub.chunks(**kw)
>>>     chunked = list(chunk_iter)
>>>     chunk_lens = list(map(len, chunked))
>>>     row = ub.dict_union(grid_item, {'chunk_lens': chunk_lens, 'chunks': chunked}
↪)
>>>     row['chunker'] = str(row['chunker'])
>>>     if not row['legacy'] and 'nchunks' in kw:
>>>         assert kw['nchunks'] == row['nchunks']
>>>     row.update(chunk_iter.__dict__)
>>>     rows.append(row)
>>> # xdoctest: +SKIP
>>> import pandas as pd
>>> df = pd.DataFrame(rows)
>>> for _, subdf in df.groupby('chunker'):
>>>     print(subdf)
```

static noborder(*items*, *chunksize*)

static cycle(*items*, *chunksize*)

static replicate(*items*, *chunksize*)

ubelt.util_list.**compress**(*items*, *flags*)

Selects from `items` where the corresponding value in `flags` is True.

Parameters

- **items** (*Iterable[Any]*) – a sequence to select items from

- **flags** (*Iterable[bool]*) – corresponding sequence of bools

Returns a subset of masked items

Return type Iterable[Any]

Notes

This function is based on `numpy.compress()`, but is pure Python and swaps the condition and array argument to be consistent with *`ubelt.take()`*.

This is equivalent to `itertools.compress()`.

Example

```
>>> import ubelt as ub
>>> items = [1, 2, 3, 4, 5]
>>> flags = [False, True, True, False, True]
>>> list(ub.compress(items, flags))
[2, 3, 5]
```

`ubelt.util_list.`**`flatten`**(*nested*)

Transforms a nested iterable into a flat iterable.

Parameters **nested** (*Iterable[Iterable[Any]]*) – list of lists

Returns flattened items

Return type Iterable[Any]

Notes

Equivalent to `more_itertools.flatten()` and `itertools.chain.from_iterable()`.

Example

```
>>> import ubelt as ub
>>> nested = [['a', 'b'], ['c', 'd']]
>>> list(ub.flatten(nested))
['a', 'b', 'c', 'd']
```

`ubelt.util_list.`**`iter_window`**(*iterable*, *size=2*, *step=1*, *wrap=False*)

Iterates through iterable with a window size. This is essentially a 1D sliding window.

Parameters

- **iterable** (*Iterable[T]*) – an iterable sequence
- **size** (*int, default=2*) – sliding window size
- **step** (*int, default=1*) – sliding step size
- **wrap** (*bool, default=False*) – wraparound flag

Returns returns a possibly overlapping windows in a sequence

Return type Iterable[T]

Notes

Similar to `more_itertools.windowed()`, Similar to `more_itertools.pairwise()`, Similar to `more_itertools.triplewise()`, Similar to `more_itertools.sliding_window()`

Example

```
>>> import ubelt as ub
>>> iterable = [1, 2, 3, 4, 5, 6]
>>> size, step, wrap = 3, 1, True
>>> window_iter = ub.iter_window(iterable, size, step, wrap)
>>> window_list = list(window_iter)
>>> print('window_list = %r' % (window_list,))
window_list = [(1, 2, 3), (2, 3, 4), (3, 4, 5), (4, 5, 6), (5, 6, 1), (6, 1, 2)]
```

Example

```
>>> import ubelt as ub
>>> iterable = [1, 2, 3, 4, 5, 6]
>>> size, step, wrap = 3, 2, True
>>> window_iter = ub.iter_window(iterable, size, step, wrap)
>>> window_list = list(window_iter)
>>> print('window_list = {!r}'.format(window_list))
window_list = [(1, 2, 3), (3, 4, 5), (5, 6, 1)]
```

Example

```
>>> import ubelt as ub
>>> iterable = [1, 2, 3, 4, 5, 6]
>>> size, step, wrap = 3, 2, False
>>> window_iter = ub.iter_window(iterable, size, step, wrap)
>>> window_list = list(window_iter)
>>> print('window_list = {!r}'.format(window_list))
window_list = [(1, 2, 3), (3, 4, 5)]
```

Example

```
>>> import ubelt as ub
>>> iterable = []
>>> size, step, wrap = 3, 2, False
>>> window_iter = ub.iter_window(iterable, size, step, wrap)
>>> window_list = list(window_iter)
>>> print('window_list = {!r}'.format(window_list))
window_list = []
```

`ubelt.util_list.`**`iterable`**(*obj*, *strok=False*)

Checks if the input implements the iterator interface. An exception is made for strings, which return False unless `strok` is True

Parameters

- **obj** (*object*) – a scalar or iterable input
- **strok** (*bool, default=False*) – if True allow strings to be interpreted as iterable

Returns True if the input is iterable

Return type bool

Example

```
>>> import ubelt as ub
>>> obj_list = [3, [3], '3', (3,), [3, 4, 5], {}]
>>> result = [ub.iterable(obj) for obj in obj_list]
>>> assert result == [False, True, False, True, True, True]
>>> result = [ub.iterable(obj, strok=True) for obj in obj_list]
>>> assert result == [False, True, True, True, True, True]
```

`ubelt.util_list.`**`peek`**(*iterable*, *default=NoParam*)

Look at the first item of an iterable. If the input is an iterator, then the next element is exhausted (i.e. a pop operation).

Parameters

- **iterable** (*Iterable[T]*) – an iterable
- **default** (*T*) – default item to return if the iterable is empty, otherwise a StopIteration error is raised

Returns

item - the first item of ordered sequence, a popped item from an iterator, or an arbitrary item from an unordered collection.

Return type T

Notes

Similar to `more_itertools.peekable()`

Example

```
>>> import ubelt as ub
>>> data = [0, 1, 2]
>>> ub.peek(data)
0
>>> iterator = iter(data)
>>> print(ub.peek(iterator))
0
>>> print(ub.peek(iterator))
1
>>> print(ub.peek(iterator))
2
>>> ub.peek(range(3))
```

(continues on next page)

(continued from previous page)

```
0
>>> ub.peek([], 3)
3
```

ubelt.util_list.**take**(*items*, *indices*, *default=NoParam*)

Lookup a subset of an indexable object using a sequence of indices.

The `items` input is usually a list or dictionary. When `items` is a list, this should be a sequence of integers. When `items` is a dict, this is a list of keys to lookup in that dictionary.

For dictionaries, a default may be specified as a placeholder to use if a key from `indices` is not in `items`.

Parameters

- **items** (*Sequence[VT] | Mapping[KT, VT]*) – An indexable object to select items from.
- **indices** (*Iterable[int | KT]*) – A sequence of indexes into `items`.
- **default** (*Any, default=NoParam*) – if specified `items` must support the `get` method.

Yields *VT* – a selected item within the list

SeeAlso: *ubelt.dict_subset()*

Note: `ub.take(items, indices)` is equivalent to `(items[i] for i in indices)` when `default` is unspecified.

Notes

This is based on the `numpy.take()` function, but written in pure python.

Do not confuse this with `more_itertools.take()`, the behavior is very different.

Example

```
>>> import ubelt as ub
>>> items = [0, 1, 2, 3]
>>> indices = [2, 0]
>>> list(ub.take(items, indices))
[2, 0]
```

Example

```
>>> import ubelt as ub
>>> dict_ = {1: 'a', 2: 'b', 3: 'c'}
>>> keys = [1, 2, 3, 4, 5]
>>> result = list(ub.take(dict_, keys, None))
>>> assert result == ['a', 'b', 'c', None, None]
```

Example

```
>>> import ubelt as ub
>>> dict_ = {1: 'a', 2: 'b', 3: 'c'}
>>> keys = [1, 2, 3, 4, 5]
>>> try:
>>>     print(list(ub.take(dict_, keys)))
>>>     raise AssertionError('did not get key error')
>>> except KeyError:
>>>     print('correctly got key error')
```

ubelt.util_list.**unique**(*items*, *key=None*)

Generates unique items in the order they appear.

Parameters

- **items** (*Iterable[T]*) – list of items
- **key** (*Callable[[T], Any], default=None*) – custom normalization function. If specified returns items where `key(item)` is unique.

Yields *T* – a unique item from the input sequence

Notes

Functionally equivalent to `more_itertools.unique_everseen()`.

Example

```
>>> import ubelt as ub
>>> items = [4, 6, 6, 0, 6, 1, 0, 2, 2, 1]
>>> unique_items = list(ub.unique(items))
>>> assert unique_items == [4, 6, 0, 1, 2]
```

Example

```
>>> import ubelt as ub
>>> items = ['A', 'a', 'b', 'B', 'C', 'c', 'D', 'e', 'D', 'E']
>>> unique_items = list(ub.unique(items, key=str.lower))
>>> assert unique_items == ['A', 'b', 'C', 'D', 'e']
>>> unique_items = list(ub.unique(items))
>>> assert unique_items == ['A', 'a', 'b', 'B', 'C', 'c', 'D', 'e', 'E']
```

ubelt.util_list.**unique_flags**(*items*, *key=None*)

Returns a list of booleans corresponding to the first instance of each unique item.

Parameters

- **items** (*Sequence[VT]*) – indexable collection of items
- **key** (*Callable[[VT], Any] | None, default=None*) – custom normalization function. If specified returns items where `key(item)` is unique.

Returns flags the items that are unique

Return type List[bool]

Example

```
>>> import ubelt as ub
>>> items = [0, 2, 1, 1, 0, 9, 2]
>>> flags = ub.unique_flags(items)
>>> assert flags == [True, True, True, False, False, True, False]
>>> flags = ub.unique_flags(items, key=lambda x: x % 2 == 0)
>>> assert flags == [True, False, True, False, False, False, False]
```

1.27.1.1.1.20 ubelt.util_memoize module

This module exposes decorators for in-memory caching of functional results. This is particularly useful when prototyping dynamic programming algorithms.

Either *memoize()*, *memoize_method()*, and *memoize_property()* should be used depending on what type of function is being wrapped. The following example demonstrates this.

In Python 3.8+ *memoize()* and *memoize_method()* work similarly to the standard library `functools.cache()` and `functools.memoize_method()`, but the ubelt version makes use of *ubelt.util_hash.hash_data()*, which is slower, but handles inputs containing mutable containers.

Example

```
>>> import ubelt as ub
>>> # Memoize a function, the args are hashed
>>> @ub.memoize
>>> def func(a, b):
>>>     return a + b
>>> #
>>> class MyClass:
>>>     # Memoize a class method, the args are hashed
>>>     @ub.memoize_method
>>>     def my_method(self, a, b):
>>>         return a + b
>>>     #
>>>     # Memoize a property: there can be no args,
>>>     @ub.memoize_property
>>>     @property
>>>     def my_property1(self):
>>>         return 4
>>>     #
>>>     # The property decorator is optional
>>>     def my_property2(self):
>>>         return 5
>>> #
>>> func(1, 2)
>>> func(1, 2)
>>> self = MyClass()
>>> self.my_method(1, 2)
```

(continues on next page)

(continued from previous page)

```
>>> self.my_method(1, 2)
>>> self.my_property1
>>> self.my_property1
>>> self.my_property2
>>> self.my_property2
```

ubelt.util_memoize.**memoize**(*func*)

memoization decorator that respects args and kwargs

In Python 3.9. The `functools` introduces the *cache* method, which is currently faster than memoize for simple functions [FunctoolsCache]. However, memoize can handle more general non-natively hashable inputs.

Parameters **func** (*Callable*) – live python function

Returns memoized wrapper

Return type Callable

References

Example

```
>>> import ubelt as ub
>>> closure = {'a': 'b', 'c': 'd'}
>>> incr = [0]
>>> def foo(key):
>>>     value = closure[key]
>>>     incr[0] += 1
>>>     return value
>>> foo_memo = ub.memoize(foo)
>>> assert foo('a') == 'b' and foo('c') == 'd'
>>> assert incr[0] == 2
>>> print('Call memoized version')
>>> assert foo_memo('a') == 'b' and foo_memo('c') == 'd'
>>> assert incr[0] == 4
>>> assert foo_memo('a') == 'b' and foo_memo('c') == 'd'
>>> print('Counter should no longer increase')
>>> assert incr[0] == 4
>>> print('Closure changes result without memoization')
>>> closure = {'a': 0, 'c': 1}
>>> assert foo('a') == 0 and foo('c') == 1
>>> assert incr[0] == 6
>>> assert foo_memo('a') == 'b' and foo_memo('c') == 'd'
```

class ubelt.util_memoize.**memoize_method**(*func*)

Bases: `object`

memoization decorator for a method that respects args and kwargs

References

Example

```
>>> import ubelt as ub
>>> closure = {'a': 'b', 'c': 'd'}
>>> incr = [0]
>>> class Foo(object):
>>>     @memoize_method
>>>     def foo_memo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>>     def foo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>> self = Foo()
>>> assert self.foo('a') == 'b' and self.foo('c') == 'd'
>>> assert incr[0] == 2
>>> print('Call memoized version')
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> assert incr[0] == 4
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Counter should no longer increase')
>>> assert incr[0] == 4
>>> print('Closure changes result without memoization')
>>> closure = {'a': 0, 'c': 1}
>>> assert self.foo('a') == 0 and self.foo('c') == 1
>>> assert incr[0] == 6
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Constructing a new object should get a new cache')
>>> self2 = Foo()
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
```

`ubelt.util_memoize.`**`memoize_property`**(*fget*)

Return a property attribute for new-style classes that only calls its getter on the first access. The result is stored and on subsequent accesses is returned, preventing the need to call the getter any more.

This decorator can either be used by itself or by decorating another property. In either case the method will always become a property.

Note: implementation is a modified version of [estebistec_memoize].

References

Example

```
>>> import ubelt as ub
>>> class C(object):
...     load_name_count = 0
...     @ub.memoize_property
...     def name(self):
...         "name's docstring"
...         self.load_name_count += 1
...         return "the name"
...     @ub.memoize_property
...     @property
...     def another_name(self):
...         "name's docstring"
...         self.load_name_count += 1
...         return "the name"
>>> c = C()
>>> c.load_name_count
0
>>> c.name
'the name'
>>> c.load_name_count
1
>>> c.name
'the name'
>>> c.load_name_count
1
>>> c.another_name
```

1.27.1.1.1.21 ubelt.util_mixins module

This module defines the *NiceRepr* mixin class, which defines a `__repr__` and `__str__` method that only depend on a custom `__nice__` method, which you must define. This means you only have to overload one function instead of two. Furthermore, if the object defines a `__len__` method, then the `__nice__` method defaults to something sensible, otherwise it is treated as abstract and raises `NotImplementedError`.

To use simply have your object inherit from *NiceRepr* (multi-inheritance should be ok).

Example

```
>>> # Objects that define __nice__ have a default __str__ and __repr__
>>> import ubelt as ub
>>> class Student(ub.NiceRepr):
...    def __init__(self, name):
...        self.name = name
...    def __nice__(self):
...        return self.name
>>> s1 = Student('Alice')
>>> s2 = Student('Bob')
```

(continues on next page)

(continued from previous page)

```
>>> # The __str__ representation looks nice
>>> print('s1 = {}'.format(s1))
>>> print('s2 = {}'.format(s2))
s1 = <Student(Alice)>
s2 = <Student(Bob)>
>>> # xdoctest: +IGNORE_WANT
>>> # The __repr__ representation also looks nice
>>> print('s1 = {!r}'.format(s1))
>>> print('s2 = {!r}'.format(s2))
s1 = <Student(Alice) at 0x7f2c5460aad0>
s2 = <Student(Bob) at 0x7f2c5460ad10>
```

Example

```
>>> # Objects that define __len__ have a default __nice__
>>> import ubelt as ub
>>> class Group(ub.NiceRepr):
...     def __init__(self, data):
...         self.data = data
...     def __len__(self):
...         return len(self.data)
>>> g = Group([1, 2, 3])
>>> print('g = {}'.format(g))
g = <Group(3)>
```

class ubelt.util_mixins.**NiceRepr**

Bases: object

Inherit from this class and define `__nice__` to "nicely" print your objects.

Defines `__str__` and `__repr__` in terms of `__nice__` function Classes that inherit from *NiceRepr* should redefine `__nice__`. If the inheriting class has a `__len__`, method then the default `__nice__` method will return its length.

Example

```
>>> import ubelt as ub
>>> class Foo(ub.NiceRepr):
...     def __nice__(self):
...         return 'info'
>>> foo = Foo()
>>> assert str(foo) == '<Foo(info)>'
>>> assert repr(foo).startswith('<Foo(info) at ')
```

Example

```
>>> import ubelt as ub
>>> class Bar(ub.NiceRepr):
...     pass
>>> bar = Bar()
>>> import pytest
>>> with pytest.warns(RuntimeWarning) as record:
>>>     assert 'object at' in str(bar)
>>>     assert 'object at' in repr(bar)
```

Example

```
>>> import ubelt as ub
>>> class Baz(ub.NiceRepr):
...     def __len__(self):
...         return 5
>>> baz = Baz()
>>> assert str(baz) == '<Baz(5)>'
```

Example

```
>>> import ubelt as ub
>>> # If your nice message has a bug, it shouldn't bring down the house
>>> class Foo(ub.NiceRepr):
...     def __nice__(self):
...         assert False
>>> foo = Foo()
>>> import pytest
>>> with pytest.warns(RuntimeWarning) as record:
>>>     print('foo = {!r}'.format(foo))
foo = <...Foo ...>
```

1.27.1.1.1.22 ubelt.util_path module

Functions for working with filesystem paths.

The *expandpath()* function expands the tilde to $HOME and environment variables to their values.

The *augpath()* function creates variants of an existing path without having to spend multiple lines of code splitting it up and stitching it back together.

The *shrinkuser()* function replaces your home directory with a tilde.

The *userhome()* function reports the home directory of the current user of the operating system.

The *ensuredir()* function operates like `mkdir -p` in unix.

The *Path* object is an extension of `pathlib.Path` that contains extra convenience methods corresponding to the extra functional methods in this module.

class ubelt.util_path.**Path**(**args*, ***kwargs*)

Bases: pathlib.PosixPath

An extension of pathlib.Path with extra convenience methods

Note:

New methods are:

- augment
- ensuredir
- expand
- expandvars
- ls
- shrinkuser
- walk

New classmethods are:

- appdir

Modified methods are:

- touch

Example

```
>>> # Ubelt extends pathlib functionality
>>> import ubelt as ub
>>> dpath = ub.Path('~/.cache/ubelt/demo_path').expand().ensuredir()
>>> fpath = dpath / 'text_file.txt'
>>> aug_fpath = fpath.augment(suffix='.aux', ext='.jpg').touch()
>>> aug_dpath = dpath.augment('demo_path2')
>>> assert aug_fpath.read_text() == ''
>>> fpath.write_text('text data')
>>> assert aug_fpath.exists()
>>> assert not aug_fpath.delete().exists()
>>> assert dpath.exists()
>>> assert not dpath.delete().exists()
>>> print(f'{str(fpath.shrinkuser()).replace(os.path.sep, "/")}')
>>> print(f'{str(dpath.shrinkuser()).replace(os.path.sep, "/")}')
>>> print(f'{str(aug_fpath.shrinkuser()).replace(os.path.sep, "/")}')
>>> print(f'{str(aug_dpath.shrinkuser()).replace(os.path.sep, "/")}')
~/.cache/ubelt/demo_path/text_file.txt
~/.cache/ubelt/demo_path
~/.cache/ubelt/demo_path/text_file.aux.jpg
~/.cache/ubelt/demo_pathdemo_path2
```

classmethod appdir(*appname*, **args*, *type='cache'*)

Returns an operating system appropriate writable directory for an application to be used for cache, configs, or data.

Parameters

- **appname** (*str*) – the name of the application
- ***args[str]** – optional subdirs
- **type** (*str*) – can be 'cache', 'config', or 'data'.

Returns a new path object

Return type *Path*

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> import ubelt as ub
>>> print(ub.Path.appdir('ubelt', type='cache').shrinkuser())
>>> print(ub.Path.appdir('ubelt', type='config').shrinkuser())
>>> print(ub.Path.appdir('ubelt', type='data').shrinkuser())
~/.cache/ubelt
~/.config/ubelt
~/.local/share/ubelt
>>> import pytest
>>> with pytest.raises(KeyError):
>>>     ub.Path.appdir('ubelt', type='other')
```

augment(*suffix=''*, *prefix=''*, *ext=None*, *stem=None*, *dpath=None*, *tail=''*, *relative=None*, *multidot=False*)

Create a new path with a different extension, basename, directory, prefix, and/or suffix.

See *augpath()* for more details.

Parameters

- **suffix** (*str*) – Text placed between the stem and extension. Default to ''.
- **prefix** (*str*) – Text placed in front of the stem. Defaults to ''.
- **ext** (*str | None*) – If specified, replaces the extension
- **stem** (*str | None*) – If specified, replaces the stem (i.e. basename without extension). Note: named base in *augpath()*.
- **dpath** (*str | PathLike | None*) – If specified, replaces the specified "relative" directory, which by default is the parent directory.
- **tail** (*str | None*) – If specified, appends this text to the extension.
- **relative** (*str | PathLike | None*) – Replaces `relative` with `dpath` in `path`. Has no effect if `dpath` is not specified. Defaults to the dirname of the input `path`. *experimental* not currently implemented.
- **multidot** (*bool*) – Allows extensions to contain multiple dots. Specifically, if False, everything after the last dot in the basename is the extension. If True, everything after the first dot in the basename is the extension.

Returns augmented path

Return type *Path*

Example

```
>>> import ubelt as ub
>>> path = ub.Path('foo.bar')
>>> suffix = '_suff'
>>> prefix = 'pref_'
>>> ext = '.baz'
>>> newpath = path.augment(suffix, prefix, ext=ext, stem='bar')
>>> print('newpath = {!r}'.format(newpath))
newpath = Path('pref_bar_suff.baz')
```

delete()

Removes a file or recursively removes a directory. If a path does not exist, then this is does nothing.

SeeAlso: *ubelt.delete()*

Returns reference to self

Return type *Path*

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> base = ub.Path(ub.ensure_app_cache_dir('ubelt', 'delete_test2'))
>>> dpath1 = (base / 'dir').ensuredir()
>>> (base / 'dir' / 'subdir').ensuredir()
>>> (base / 'dir' / 'to_remove1.txt').touch()
>>> fpath1 = (base / 'dir' / 'subdir' / 'to_remove3.txt').touch()
>>> fpath2 = (base / 'dir' / 'subdir' / 'to_remove2.txt').touch()
>>> assert all(p.exists() for p in [dpath1, fpath1, fpath2])
>>> fpath1.delete()
>>> assert all(p.exists() for p in [dpath1, fpath2])
>>> assert not fpath1.exists()
>>> dpath1.delete()
>>> assert not any(p.exists() for p in [dpath1, fpath1, fpath2])
```

ensuredir(*mode=511*)

Concise alias of `self.mkdir(parents=True, exist_ok=True)`

Returns returns itself

Return type *Path*

Example

```
>>> import ubelt as ub
>>> cache_dpath = ub.ensure_app_cache_dir('ubelt')
>>> dpath = ub.Path(join(cache_dpath, 'ensuredir'))
>>> if dpath.exists():
...     os.rmdir(dpath)
>>> assert not dpath.exists()
>>> dpath.ensuredir()
>>> assert dpath.exists()
>>> dpath.rmdir()
```

expand()

Expands user tilde and environment variables.

Concise alias of *Path(os.path.expandvars(self.expanduser()))*

Returns path with expanded environment variables and tildes

Return type *Path*

Example

```
>>> import ubelt as ub
>>> #home_v1 = ub.Path('$HOME').expand()
>>> home_v2 = ub.Path('~/').expand()
>>> assert isinstance(home_v2, ub.Path)
>>> home_v3 = ub.Path.home()
>>> #print('home_v1 = {!r}'.format(home_v1))
>>> print('home_v2 = {!r}'.format(home_v2))
>>> print('home_v3 = {!r}'.format(home_v3))
>>> assert home_v3 == home_v2 # == home_v1
```

expandvars()

As discussed in [CPythonIssue21301], CPython won't be adding expandvars to pathlib. I think this is a mistake, so I added it in this extension.

Returns path with expanded environment variables

Return type *Path*

References

ls()

A convenience function to list all paths in a directory.

This is simply a wraper around iterdir that returns the results as a list instead of a generator. This is mainly for faster navigation in IPython. In production code *iterdir* should be used instead.

Returns List[Path]

Example

```
>>> import ubelt as ub
>>> self = ub.Path.appdir('ubelt/tests/ls')
>>> (self / 'dir1').ensuredir()
>>> (self / 'dir2').ensuredir()
>>> (self / 'file1').touch()
>>> (self / 'file2').touch()
>>> (self / 'dir1/file3').touch()
>>> (self / 'dir2/file4').touch()
>>> children = self.ls()
>>> assert isinstance(children, list)
>>> print(ub.repr2(sorted([p.relative_to(self) for p in children])))
[
    Path('dir1'),
    Path('dir2'),
    Path('file1'),
    Path('file2'),
]
```

shrinkuser(*home='~'*)

Inverse of `os.path.expanduser()`.

Parameters **home** (*str*) – symbol used to replace the home path. Defaults to '~', but you might want to use '$HOME' or '%USERPROFILE%' instead.

Returns path - shortened path replacing the home directory with a symbol

Return type *Path*

Example

```
>>> import ubelt as ub
>>> path = ub.Path('~').expand()
>>> assert str(path.shrinkuser()) == '~'
>>> assert str(ub.Path((str(path) + '1')).shrinkuser()) == str(path) + '1'
>>> assert str((path / '1').shrinkuser()) == join('~', '1')
>>> assert str((path / '1').shrinkuser('$HOME')) == join('$HOME', '1')
>>> assert str(ub.Path('.').shrinkuser()) == '.'
```

touch(*mode=438*, *exist_ok=True*)

Create this file with the given access mode, if it doesn't exist.

Returns returns itself

Return type *Path*

Notes

The `ubelt.util_io.touch()` function currently has a slightly different implementation. This uses whatever the pathlib version is. This may change in the future.

walk(*topdown=True*, *onerror=None*, *followlinks=False*)

A variant of os.walk for pathlib

Parameters

- **topdown** (*bool*) – if True starts yield nodes closer to the root first otherwise yield nodes closer to the leaves first.
- **onerror** (*Callable[[OSError], None]*) – A function with one argument of type OSError. If the error is raised the walk is aborted, otherwise it continues.
- **followlinks** (*bool*) – if True recurse into symbolic directory links

Yields *Tuple['Path', str, str]* – the root, file names, and directory names

Example

```
>>> import ubelt as ub
>>> self = ub.Path.appdir('ubelt/tests/ls')
>>> (self / 'dir1').ensuredir()
>>> (self / 'dir2').ensuredir()
>>> (self / 'file1').touch()
>>> (self / 'file2').touch()
>>> (self / 'dir1/file3').touch()
>>> (self / 'dir2/file4').touch()
>>> subdirs = list(self.walk())
>>> assert len(subdirs) == 3
```

class ubelt.util_path.**TempDir**

Bases: `object`

Context for creating and cleaning up temporary directories.

Note: This class will be DEPRECATED. The exact deprecation version and mitigation plan has not yet been developed.

Note: This exists because `tempfile.TemporaryDirectory` was introduced in Python 3.2. Thus once ubelt no longer supports python 2.7, this class will be deprecated.

Example

```
>>> from ubelt.util_path import *  # NOQA
>>> with TempDir() as self:
>>>     dpath = self.dpath
>>>     assert exists(dpath)
>>> assert not exists(dpath)
```

Example

```
>>> from ubelt.util_path import *  # NOQA
>>> self = TempDir()
>>> dpath = self.ensure()
>>> assert exists(dpath)
>>> self.cleanup()
>>> assert not exists(dpath)
```

ensure()

cleanup()

start()

ubelt.util_path.**augpath**(*path*, *suffix=''*, *prefix=''*, *ext=None*, *tail=''*, *base=None*, *dpath=None*, *relative=None*, *multidot=False*)

Create a new path with a different extension, basename, directory, prefix, and/or suffix.

A prefix is inserted before the basename. A suffix is inserted between the basename and the extension. The basename and extension can be replaced with a new one. Essentially a path is broken down into components (dpath, base, ext), and then recombined as (dpath, prefix, base, suffix, ext) after replacing any specified component.

Parameters

- **path** (*str* | *PathLike*) – a path to augment
- **suffix** (*str*) – placed between the basename and extension
- **prefix** (*str*) – placed in front of the basename
- **ext** (*str* | *None*) – if specified, replaces the extension
- **tail** (*str* | *None*) – If specified, appends this text to the extension
- **base** (*str* | *None*) – if specified, replaces the basename without extension. Note: this is referred to as stem in `ub.Path`.
- **dpath** (*str* | *PathLike* | *None*) – if specified, replaces the specified "relative" directory, which by default is the parent directory.
- **relative** (*str* | *PathLike* | *None*) – Replaces `relative` with `dpath` in `path`. Has no effect if `dpath` is not specified. Defaults to the dirname of the input `path`. *experimental* not currently implemented.
- **multidot** (*bool*) – Allows extensions to contain multiple dots. Specifically, if False, everything after the last dot in the basename is the extension. If True, everything after the first dot in the basename is the extension.

Returns augmented path

Return type str

Example

```
>>> import ubelt as ub
>>> path = 'foo.bar'
>>> suffix = '_suff'
>>> prefix = 'pref_'
>>> ext = '.baz'
>>> newpath = ub.augpath(path, suffix, prefix, ext=ext, base='bar')
>>> print('newpath = %s' % (newpath,))
newpath = pref_bar_suff.baz
```

Example

```
>>> from ubelt.util_path import *  # NOQA
>>> augpath('foo.bar')
'foo.bar'
>>> augpath('foo.bar', ext='.BAZ')
'foo.BAZ'
>>> augpath('foo.bar', suffix='_')
'foo_.bar'
>>> augpath('foo.bar', prefix='_')
'_foo.bar'
>>> augpath('foo.bar', base='baz')
'baz.bar'
>>> augpath('foo.tar.gz', ext='.zip', multidot=True)
foo.zip
>>> augpath('foo.tar.gz', ext='.zip', multidot=False)
foo.tar.zip
>>> augpath('foo.tar.gz', suffix='_new', multidot=True)
foo_new.tar.gz
>>> augpath('foo.tar.gz', suffix='_new', tail='.cache', multidot=True)
foo_new.tar.gz.cache
```

`ubelt.util_path.`**`shrinkuser`**(*path*, *home='~'*)

Inverse of `os.path.expanduser()`.

Parameters

- **path** (*str* | *PathLike*) – path in system file structure
- **home** (*str*) – symbol used to replace the home path. Defaults to '~', but you might want to use '$HOME' or '%USERPROFILE%' instead.

Returns path - shortened path replacing the home directory with a symbol

Return type str

Example

```
>>> from ubelt.util_path import *  # NOQA
>>> path = expanduser('~')
>>> assert path != '~'
>>> assert shrinkuser(path) == '~'
>>> assert shrinkuser(path + '1') == path + '1'
>>> assert shrinkuser(path + '/1') == join('~', '1')
>>> assert shrinkuser(path + '/1', '$HOME') == join('$HOME', '1')
>>> assert shrinkuser('.') == '.'
```

ubelt.util_path.**userhome**(*username=None*)

Returns the path to some user's home directory.

Parameters **username** (*str | None*) – name of a user on the system. If not specified, the current user is inferred.

Returns userhome_dpath - path to the specified home directory

Return type str

Raises

- KeyError – if the specified user does not exist on the system
- OSError – if username is unspecified and the current user cannot be inferred

Example

```
>>> from ubelt.util_path import *  # NOQA
>>> import getpass
>>> username = getpass.getuser()
>>> assert userhome() == expanduser('~')
>>> assert userhome(username) == expanduser('~')
```

ubelt.util_path.**ensuredir**(*dpath, mode=1023, verbose=0, recreate=False*)

Ensures that directory will exist. Creates new dir with sticky bits by default

Parameters

- **dpath** (*str | PathLike | Tuple[str | PathLike]*) – dir to ensure. Can also be a tuple to send to join
- **mode** (*int*) – octal mode of directory
- **verbose** (*int*) – verbosity
- **recreate** (*bool*) – if True removes the directory and all of its contents and creates a fresh new directory. USE CAREFULLY.

Returns path - the ensured directory

Return type str

SeeAlso: *ubelt.Path.ensuredir()*

Note: This function is not thread-safe in Python2

Example

```
>>> from ubelt.util_path import *  # NOQA
>>> import ubelt as ub
>>> cache_dpath = ub.ensure_app_cache_dir('ubelt')
>>> dpath = join(cache_dpath, 'ensuredir')
>>> if exists(dpath):
...     os.rmdir(dpath)
>>> assert not exists(dpath)
>>> ub.ensuredir(dpath)
>>> assert exists(dpath)
>>> os.rmdir(dpath)
```

ubelt.util_path.**expandpath**(*path*)

Shell-like environment variable and tilde path expansion.

Parameters **path** (*str* | *PathLike*) – string representation of a path

Returns expanded path

Return type str

Example

```
>>> from ubelt.util_path import *  # NOQA
>>> import ubelt as ub
>>> assert normpath(ub.expandpath('~/foo')) == join(ub.userhome(), 'foo')
>>> assert ub.expandpath('foo') == 'foo'
```

1.27.1.1.1.23 ubelt.util_platform module

The goal of this module is to provide an idiomatic cross-platform pattern of accessing platform dependent file systems.

Standard application directory structure: cache, config, and other XDG standards [XDG_Spec]. This is similar to the more focused `appdirs` module [AS_appdirs]. In the future ubelt may directly use `appdirs`.

Note: Table mapping the type of directory to the system default environment variable. Inspired by [SO_43853548], [SO_11113974], and [harawata_appdirs].

```
       | Linux            | Win32          |   Darwin
data   | $XDG_DATA_HOME   | %APPDATA%      | ~/Library/Application Support
config | $XDG_CONFIG_HOME | %APPDATA%      | ~/Library/Application Support
cache  | $XDG_CACHE_HOME  | %LOCALAPPDATA% | ~/Library/Caches


If an environment variable is not specified the defaults are:
    APPDATA      = ~/AppData/Roaming
    LOCALAPPDATA = ~/AppData/Local

    XDG_DATA_HOME   = ~/.local/share
```

(continues on next page)

(continued from previous page)

```
    XDG_CACHE_HOME  = ~/.cache
    XDG_CONFIG_HOME = ~/.config
```

References

`ubelt.util_platform.find_exe(name, multi=False, path=None)`

Locate a command.

Search your local filesystem for an executable and return the first matching file with executable permission.

Parameters

- **name** (*str | PathLike*) – globstr of matching filename
- **multi** (*bool, default=False*) – if True return all matches instead of just the first.
- **path** (*str | PathLike | Iterable[str | PathLike] | None, default=None*) – overrides the system PATH variable.

Returns returns matching executable(s).

Return type str | List[str] | None

SeeAlso: `shutil.which()` - which is available in Python 3.3+.

Note: This is essentially the `which` UNIX command

References

Example

```
>>> find_exe('ls')
>>> find_exe('ping')
>>> assert find_exe('which') == find_exe(find_exe('which'))
>>> find_exe('which', multi=True)
>>> find_exe('ping', multi=True)
>>> find_exe('cmake', multi=True)
>>> find_exe('nvcc', multi=True)
>>> find_exe('noexist', multi=True)
```

Example

```
>>> import ubelt as ub
>>> assert not ub.find_exe('noexist', multi=False)
>>> assert ub.find_exe('ping', multi=False) or ub.find_exe('ls', multi=False)
>>> assert not ub.find_exe('noexist', multi=True)
>>> assert ub.find_exe('ping', multi=True) or ub.find_exe('ls', multi=True)
```

Benchmark:

```
>>> # xdoctest: +IGNORE_WANT
>>> import ubelt as ub
>>> import shutil
>>> for timer in ub.Timerit(100, bestof=10, label='ub.find_exe'):
>>>     ub.find_exe('which')
>>> for timer in ub.Timerit(100, bestof=10, label='shutil.which'):
>>>     shutil.which('which')
Timed best=58.71 µs, mean=59.64 ± 0.96 µs for ub.find_exe
Timed best=72.75 µs, mean=73.07 ± 0.22 µs for shutil.which
```

`ubelt.util_platform.`**`find_path`**(*name*, *path=None*, *exact=False*)

Search for a file or directory on your local filesystem by name (file must be in a directory specified in a PATH environment variable)

Parameters

- **name** (*str | PathLike*) – file name to match. If exact is False this may be a glob pattern
- **path** (*str | Iterable[str | PathLike], default=None*) – list of directories to search either specified as an `os.pathsep` separated string or a list of directories. Defaults to environment PATH.
- **exact** (*bool, default=False*) – if True, only returns exact matches.

Yields *str* – candidate - a path that matches `name`

Note: Running with `name=''` (i.e. `ub.find_path('')`) will simply yield all directories in your PATH.

Note: For recursive behavior set `path=(d for d, _, _ in os.walk('.'))`, where '.' might be replaced by the root directory of interest.

Example

```
>>> list(find_path('ping', exact=True))
>>> list(find_path('bin'))
>>> list(find_path('*cc*'))
>>> list(find_path('cmake*'))
```

Example

```
>>> import ubelt as ub
>>> from os.path import dirname
>>> path = dirname(dirname(ub.util_platform.__file__))
>>> res = sorted(find_path('ubelt/util_*.py', path=path))
>>> assert len(res) >= 10
>>> res = sorted(find_path('ubelt/util_platform.py', path=path, exact=True))
>>> print(res)
>>> assert len(res) == 1
```

ubelt.util_platform.**ensure_app_cache_dir**(*appname*, **args*)

Calls *get_app_cache_dir()* but ensures the directory exists.

Parameters

- **appname** (*str*) – the name of the application
- ***args** – any other subdirectories may be specified

Returns the path to the ensured directory

Return type str

SeeAlso: *get_app_cache_dir()*

Example

```
>>> import ubelt as ub
>>> dpath = ub.ensure_app_cache_dir('ubelt')
>>> assert exists(dpath)
```

ubelt.util_platform.**ensure_app_config_dir**(*appname*, **args*)

Calls *get_app_config_dir()* but ensures the directory exists.

Parameters

- **appname** (*str*) – the name of the application
- ***args** – any other subdirectories may be specified

Returns the path to the ensured directory

Return type str

SeeAlso: *get_app_config_dir()*

Example

```
>>> import ubelt as ub
>>> dpath = ub.ensure_app_config_dir('ubelt')
>>> assert exists(dpath)
```

ubelt.util_platform.**ensure_app_data_dir**(*appname*, **args*)

Calls *get_app_data_dir()* but ensures the directory exists.

Parameters

- **appname** (*str*) – the name of the application
- ***args** – any other subdirectories may be specified

Returns the path to the ensured directory

Return type str

SeeAlso: *get_app_data_dir()*

Example

```
>>> import ubelt as ub
>>> dpath = ub.ensure_app_data_dir('ubelt')
>>> assert exists(dpath)
```

ubelt.util_platform.**get_app_cache_dir**(*appname*, **args*)

Returns a writable directory for an application. This should be used for temporary deletable data.

Parameters

- **appname** (*str*) – the name of the application
- ***args** – any other subdirectories may be specified

Returns the path to the ensured directory

Return type str

Returns dpath - writable cache directory for this application

Return type str

SeeAlso: *ensure_app_cache_dir()*

ubelt.util_platform.**get_app_config_dir**(*appname*, **args*)

Returns a writable directory for an application This should be used for persistent configuration files.

Parameters

- **appname** (*str*) – the name of the application
- ***args** – any other subdirectories may be specified

Returns dpath - writable config directory for this application

Return type str

SeeAlso: *ensure_app_config_dir()*

ubelt.util_platform.**get_app_data_dir**(*appname*, **args*)

Returns a writable directory for an application. This should be used for temporary deletable data.

Parameters

- **appname** (*str*) – the name of the application
- ***args** – any other subdirectories may be specified

Returns dpath - writable data directory for this application

Return type str

SeeAlso: *ensure_app_data_dir()*

ubelt.util_platform.**platform_cache_dir**()

Returns a directory which should be writable for any application This should be used for temporary deletable data.

Returns path to the cache dir used by the current operating system

Return type str

`ubelt.util_platform.platform_config_dir()`

Returns a directory which should be writable for any application This should be used for persistent configuration files.

Returns path to the cache dir used by the current operating system

Return type str

`ubelt.util_platform.platform_data_dir()`

Returns path for user-specific data files

Returns path to the data dir used by the current operating system

Return type str

1.27.1.1.1.24 ubelt.util_str module

Functions for working with text and strings.

The *ensure_unicode()* function does its best to coerce Python 2/3 bytes and text into a consistent unicode text representation.

The *codeblock()* and *paragraph()* wrap multiline strings to help write text blocks without hindering the surrounding code indentation.

The *hzcat()* function horizontally concatenates multiline text.

The *indent()* prefixes all lines in a text block with a given prefix. By default that prefix is 4 spaces.

`ubelt.util_str.indent(text, prefix=' ')`

Indents a block of text

Parameters

- **text** (*str*) – text to indent
- **prefix** (*str, default* = ' ') – prefix to add to each line

Returns indented text

Return type str

Example

```
>>> import ubelt as ub
>>> NL = chr(10)  # newline character
>>> text = 'Lorem ipsum' + NL + 'dolor sit amet'
>>> prefix = '    '
>>> result = ub.indent(text, prefix)
>>> assert all(t.startswith(prefix) for t in result.split(NL))
```

`ubelt.util_str.codeblock(text)`

Create a block of text that preserves all newlines and relative indentation

Wraps multiline string blocks and returns unindented code. Useful for templated code defined in indented parts of code.

Parameters **text** (*str*) – typically a multiline string

Returns the unindented string

Return type str

Example

```
>>> import ubelt as ub
>>> # Simulate an indented part of code
>>> if True:
>>>     # notice the indentation on this will be normal
>>>     codeblock_version = ub.codeblock(
...             '''
...             def foo():
...                 return 'bar'
...             '''
...         )
>>>     # notice the indentation and newlines on this will be odd
>>>     normal_version = ('''
...         def foo():
...             return 'bar'
...     ''')
>>> assert normal_version != codeblock_version
>>> print('Without codeblock')
>>> print(normal_version)
>>> print('With codeblock')
>>> print(codeblock_version)
```

ubelt.util_str.**paragraph**(*text*)

Wraps multi-line strings and restructures the text to remove all newlines, heading, trailing, and double spaces.

Useful for writing log messages

Parameters **text** (*str*) – typically a multiline string

Returns the reduced text block

Return type str

Example

```
>>> import ubelt as ub
>>> text = (
>>>     '''
>>>     Lorem ipsum dolor sit amet, consectetur adipiscing
>>>     elit, sed do eiusmod tempor incididunt ut labore et
>>>     dolore magna aliqua.
>>>     ''')
>>> out = ub.paragraph(text)
>>> assert chr(10) in text
>>> assert chr(10) not in out
>>> print('text = {!r}'.format(text))
>>> print('out = {!r}'.format(out))
```

ubelt.util_str.**hzcat**(*args*, *sep=''*)

Horizontally concatenates strings preserving indentation

Concatenates a list of objects ensuring that the next item in the list is all the way to the right of any previous items.

Parameters

- **args** (*List[str]*) – strings to concatenate
- **sep** (*str, default=''*) – separator

Example1:

```
>>> import ubelt as ub
>>> B = ub.repr2([[1, 2], [3, 457]], nl=1, cbr=True, trailsep=False)
>>> C = ub.repr2([[5, 6], [7, 8]], nl=1, cbr=True, trailsep=False)
>>> args = ['A = ', B, ' * ', C]
>>> print(ub.hzcat(args))
A = [[1, 2],   * [[5, 6],
     [3, 457]]    [7, 8]]
```

Example2:

```
>>> import ubelt as ub
>>> import unicodedata
>>> aa = unicodedata.normalize('NFD', 'á')  # a unicode char with len2
>>> B = ub.repr2([['', aa], [aa, aa, aa]], nl=1, si=True, cbr=True,␣
↪trailsep=False)
>>> C = ub.repr2([[5, 6], [7, '']], nl=1, si=True, cbr=True, trailsep=False)
>>> args = ['A', '=', B, '*', C]
>>> print(ub.hzcat(args, sep=''))
A=[[, á],   *[[5, 6],
   [á, á, á]]  [7, ]]
```

`ubelt.util_str.`**`ensure_unicode`**(*text*)

Casts bytes into utf8 (mostly for python2 compatibility)

Parameters **text** (*str | bytes*) – text to ensure is decoded as unicode

Returns str

References

[SO_12561063] http://stackoverflow.com/questions/12561063/extract-data-from-file

Example

```
>>> from ubelt.util_str import *
>>> import codecs  # NOQA
>>> assert ensure_unicode('my ünicôdé strıng') == 'my ünicôdé strıng'
>>> assert ensure_unicode('text1') == 'text1'
>>> assert ensure_unicode('text1'.encode('utf8')) == 'text1'
>>> assert ensure_unicode('ï»¿text1'.encode('utf8')) == 'ï»¿text1'
>>> assert (codecs.BOM_UTF8 + 'text»¿'.encode('utf8')).decode('utf8')
```

1.27.1.1.1.25 ubelt.util_stream module

Functions for capturing and redirecting IO streams.

The *CaptureStdout* captures all text sent to stdout and optionally prevents it from actually reaching stdout.

The *TeeStringIO* does the same thing but for arbitrary streams. It is how the former is implemented.

class ubelt.util_stream.**TeeStringIO**(*redirect=None*)

Bases: _io.StringIO

An IO object that writes to itself and another IO stream.

Variables **redirect** (*io.IOBase*) – The other stream to write to.

Example

```
>>> import ubelt as ub
>>> redirect = io.StringIO()
>>> self = ub.TeeStringIO(redirect)
```

isatty()

Returns true of the redirect is a terminal.

Note: Needed for IPython.embed to work properly when this class is used to override stdout / stderr.

fileno()

Returns underlying file descriptor of the redirected IOBase object if one exists.

Example

```
>>> # Not sure the best way to test, this func is important for
>>> # capturing stdout when ipython embedding
>>> import pytest
>>> with pytest.raises(io.UnsupportedOperation):
>>>     TeeStringIO(redirect=io.StringIO()).fileno()
>>> with pytest.raises(io.UnsupportedOperation):
>>>     TeeStringIO(None).fileno()
```

property encoding

Gets the encoding of the *redirect* IO object

Example

```
>>> import ubelt as ub
>>> redirect = io.StringIO()
>>> assert ub.TeeStringIO(redirect).encoding is None
>>> assert ub.TeeStringIO(None).encoding is None
>>> assert ub.TeeStringIO(sys.stdout).encoding is sys.stdout.encoding
>>> redirect = io.TextIOWrapper(io.StringIO())
>>> assert ub.TeeStringIO(redirect).encoding is redirect.encoding
```

write(*msg*)

Write to this and the redirected stream

flush()

Flush to this and the redirected stream

class ubelt.util_stream.**CaptureStdout**(*suppress=True*, *enabled=True*)

Bases: *ubelt.util_stream.CaptureStream*

Context manager that captures stdout and stores it in an internal stream

Parameters

- **suppress** (*bool, default=True*) – if True, stdout is not printed while captured
- **enabled** (*bool, default=True*) – does nothing if this is False
- ****kwargs** – used for backwards compatibility with misspelled deprecated params.

Example

```
>>> import ubelt as ub
>>> self = ub.CaptureStdout(suppress=True)
>>> print('dont capture the table flip (°° ')
>>> with self:
...     text = 'capture the heart '
...     print(text)
>>> print('dont capture look of disapproval _')
>>> assert isinstance(self.text, str)
>>> assert self.text == text + '\n', 'failed capture text'
```

Example

```
>>> import ubelt as ub
>>> self = ub.CaptureStdout(suppress=False)
>>> with self:
...     print('I am captured and printed in stdout')
>>> assert self.text.strip() == 'I am captured and printed in stdout'
```

Example

```
>>> import ubelt as ub
>>> self = ub.CaptureStdout(suppress=True, enabled=False)
>>> with self:
...     print('dont capture')
>>> assert self.text is None
```

log_part()

Log what has been captured so far

start()

stop()

Example

```
>>> import ubelt as ub
>>> ub.CaptureStdout(enabled=False).stop()
>>> ub.CaptureStdout(enabled=True).stop()
```

close()

class ubelt.util_stream.**CaptureStream**

Bases: object

Generic class for capturing streaming output from stdout or stderr

1.27.1.1.1.26 ubelt.util_time module

This is util_time, it contains functions for handling time related code.

The *timestamp()* function returns an iso8601 timestamp without much fuss.

The *timeparse()* is the inverse of *timestamp*, and makes use of `dateutil` if it is available.

The *Timer* class is a context manager that times a block of indented code. It includes *tic* and *toc* methods a more matlab like feel.

Timerit is gone! Use the standalone and separate module **:module:`timerit`**.

See also:

`tempora` - https://github.com/jaraco/tempora - time related utility functions from Jaraco

ubelt.util_time.**timestamp**(*datetime=None*, *precision=0*, *method='iso8601'*)

Make a concise iso8601 timestamp suitable for use in filenames.

Parameters

- **datetime** (*datetime.datetime | None*) – A datetime to format into a timestamp. If unspecified, the current local time is used.
- **method** (*str*) – Type of timestamp. Currently the only option is iso8601. This argument may be removed in the future.
- **precision** (*int*) – if non-zero, adds up to 6 digits of sub-second precision.

Returns stamp

Return type str

Note: For more info see [WikiISO8601], [PyStrptime], [PyTime].

References

Todo:

- [ ] Allow defaulting to local or utm timezone (currently default is local)

Example

```
>>> import ubelt as ub
>>> stamp = ub.timestamp()
>>> print('stamp = {!r}'.format(stamp))
stamp = ...-...-...T...
```

Example

```
>>> import ubelt as ub
>>> import datetime as datetime_mod
>>> from datetime import datetime as datetime_cls
>>> # Create a datetime object with timezone information
>>> ast_tzinfo = datetime_mod.timezone(datetime_mod.timedelta(hours=-4), 'AST')
>>> datetime = datetime_cls.utcfromtimestamp(123456789.123456789).
↪replace(tzinfo=ast_tzinfo)
>>> stamp = ub.timestamp(datetime, precision=2)
>>> print('stamp = {!r}'.format(stamp))
stamp = '1973-11-29T213309.12-4'
```

```
>>> # Demo with a fractional hour timezone
>>> act_tzinfo = datetime_mod.timezone(datetime_mod.timedelta(hours=+9.5), 'ACT')
>>> datetime = datetime_cls.utcfromtimestamp(123456789.123456789).
↪replace(tzinfo=act_tzinfo)
>>> stamp = ub.timestamp(datetime, precision=2)
>>> print('stamp = {!r}'.format(stamp))
stamp = '1973-11-29T213309.12+0930'
```

Example

```
>>> # xdoctest: +REQUIRES(module:dateutil)
>>> # Make sure we are compatible with dateutil
>>> import ubelt as ub
>>> from dateutil.tz import tzlocal
>>> import datetime as datetime_mod
>>> from datetime import datetime as datetime_cls
>>> tzinfo_list = [
>>>     datetime_mod.timezone(datetime_mod.timedelta(hours=+9.5), 'ACT'),
>>>     datetime_mod.timezone(datetime_mod.timedelta(hours=-4), 'AST'),
>>>     datetime_mod.timezone(datetime_mod.timedelta(hours=0), 'UTC'),
>>>     datetime_mod.timezone.utc,
```

(continues on next page)

(continued from previous page)

```
>>>     None,
>>>     tzlocal()
>>> ]
>>> # Note: there is a win32 bug here
>>> # https://bugs.python.org/issue37 that means we cant use
>>> # dates close to the epoch
>>> datetime_list = [
>>>     datetime_cls.utcfromtimestamp(123456789.123456789 + 315360000),
>>>     datetime_cls.utcfromtimestamp(0 + 315360000),
>>> ]
>>> basis = {
>>>     'precision': [0, 3, 9],
>>>     'tzinfo': tzinfo_list,
>>>     'datetime': datetime_list,
>>> }
>>> for params in ub.named_product(basis):
>>>     dtime = params['datetime'].replace(tzinfo=params['tzinfo'])
>>>     precision = params.get('precision', 0)
>>>     stamp = ub.timestamp(datetime=dtime, precision=precision)
>>>     recon = ub.timeparse(stamp)
>>>     alt = recon.strftime('%Y-%m-%dT%H%M%S.%f%z')
>>>     print('---')
>>>     print('params = {}'.format(ub.repr2(params, nl=1)))
>>>     print(f'dtime={dtime}')
>>>     print(f'stamp={stamp}')
>>>     print(f'recon={recon}')
>>>     print(f'alt  ={alt}')
>>>     shift = 10 ** precision
>>>     a = int(dtime.timestamp() * shift)
>>>     b = int(recon.timestamp() * shift)
>>>     assert a == b, f'{a} != {b}'
```

`ubelt.util_time.`**`timeparse`**(*stamp*, *allow_dateutil=True*)

Create a `datetime.datetime` object from a string timestamp.

Without any extra dependencies this will parse the output of *`ubelt.util_time.timestamp()`* into a datetime object. In the case where the format differs, *dateutil.parser.parse* will be used if the *python-dateutil* package is installed.

Parameters

- **stamp** (*str*) – a string encoded timestamp
- **allow_dateutil** (*bool*) – if False we only use the minimal parsing and do not allow a fallback to dateutil.

Returns the parsed datetime

Return type datetime.datetime

Raises `ValueError` – if if parsing fails.

Todo:

- [ ] Allow defaulting to local or utm timezone (currently default is local)

Example

```
>>> import ubelt as ub
>>> # Demonstrate a round trip of timestamp and timeparse
>>> stamp = ub.timestamp()
>>> datetime = ub.timeparse(stamp)
>>> assert ub.timestamp(datetime) == stamp
>>> # Round trip with precision
>>> stamp = ub.timestamp(precision=4)
>>> datetime = ub.timeparse(stamp)
>>> assert ub.timestamp(datetime, precision=4) == stamp
```

Example

```
>>> import ubelt as ub
>>> # We should always be able to parse these
>>> good_stamps = [
>>>     '2000-11-22T111111.44444Z',
>>>     '2000-11-22T111111.44444+5',
>>>     '2000-11-22T111111.44444-05',
>>>     '2000-11-22T111111.44444-0500',
>>>     '2000-11-22T111111.44444+0530',
>>>     '2000-11-22T111111Z',
>>>     '2000-11-22T111111+5',
>>>     '2000-11-22T111111+0530',
>>> ]
>>> for stamp in good_stamps:
>>>     print(f'----')
>>>     print(f'stamp={stamp}')
>>>     result = ub.timeparse(stamp, allow_dateutil=0)
>>>     print(f'result={result!r}')
>>>     recon = ub.timestamp(result)
>>>     print(f'recon={recon}')
```

Example

```
>>> import ubelt as ub
>>> # We require dateutil to handle these types of stamps
>>> import pytest
>>> conditional_stamps = [
>>>         '2000-01-02T11:23:58.12345+5:30',
>>>         '09/25/2003',
>>>         'Thu Sep 25 10:36:28 2003',
>>> ]
>>> for stamp in conditional_stamps:
>>>     with pytest.raises(ValueError):
>>>         result = ub.timeparse(stamp, allow_dateutil=False)
>>> have_dateutil = bool(ub.modname_to_modpath('dateutil'))
>>> if have_dateutil:
```

(continues on next page)

(continued from previous page)

```
>>>     for stamp in conditional_stamps:
>>>         result = ub.timeparse(stamp)
```

class ubelt.util_time.**Timer**(*label=''*, *verbose=None*, *newline=True*)

Bases: object

Measures time elapsed between a start and end point. Can be used as a with-statement context manager, or using the tic/toc api.

Parameters

- **label** (*str, default=''*) – identifier for printing
- **verbose** (*int, default=None*) – verbosity flag, defaults to True if label is given
- **newline** (*bool, default=True*) – if False and verbose, print tic and toc on the same line

Variables

- **elapsed** (*float*) – number of seconds measured by the context manager
- **tstart** (*float*) – time of last *tic* reported by *self._time()*

Example

```
>>> # Create and start the timer using the context manager
>>> import math
>>> timer = Timer('Timer test!', verbose=1)
>>> with timer:
>>>     math.factorial(10)
>>> assert timer.elapsed > 0
tic('Timer test!')
...toc('Timer test!')=...
```

Example

```
>>> # Create and start the timer using the tic/toc interface
>>> timer = Timer().tic()
>>> elapsed1 = timer.toc()
>>> elapsed2 = timer.toc()
>>> elapsed3 = timer.toc()
>>> assert elapsed1 <= elapsed2
>>> assert elapsed2 <= elapsed3
```

tic()

starts the timer

toc()

stops the timer

1.27.1.1.1.27 ubelt.util_zip module

Abstractions for working with zipfiles and archives

This may be renamed to util_archive in the future.

The `ubelt.split_archive()` works with paths that reference a file inside of an archive (e.g. a zipfile). It splits it into two parts, the full path to the archive and then the path to the file inside of the archive. By convention these are separated with either a pathsep or a colon.

The `ubelt.zopen()` works to open a file that lives inside of an archive without the user needing to worry about extracting it first. When possible it will read it directly from the archive, but in some cases it may extract it to a temporary directory first.

class `ubelt.util_zip.`**`zopen`**(*fpath*, *mode='r'*, *seekable=False*, *ext='.zip'*)

Bases: `ubelt.util_mixins.NiceRepr`

An abstraction of the normal `open()` function that can also handle reading data directly inside of zipfiles.

This is a file-object like interface [FileObj] — i.e. it supports the read and write methods to an underlying resource.

Can open a file normally or open a file within a zip file (readonly). Tries to read from memory only, but will extract to a tempfile if necessary.

Just treat the zipfile like a directory, e.g. /path/to/myzip.zip/compressed/path.txt OR? e.g. /path/to/myzip.zip:compressed/path.txt

References

Todo:

- **[ ] Fast way to open a base zipfile, query what is inside, and** then choose a file to further zopen (and passing along the same open zipfile reference maybe?).
- [ ] Write mode in some restricted setting?

Parameters

- **fpath** (*str | PathLike*) – path to a file, or a special path that denotes both a path to a zipfile and a path to a archived file inside of the zipfile.
- **mode** (*str*) – Currently only "r" - readonly mode is supported
- **seekable** (*bool*) – If True, attempts to force "seekability" of the underlying file-object, for compressed files this will first extract the file to a temporary location on disk. If False, any underlying compressed file will be opened directly which may result in the object being non-seekable.
- **ext** (*str*) – The extension of the zipfile. Modify this is a non-standard extension is used (e.g. for torch packages).

Example

```
>>> from ubelt.util_zip import *  # NOQA
>>> import pickle
>>> import ubelt as ub
>>> dpath = ub.ensure_app_cache_dir('ubelt/tests/util_zip')
>>> dpath = ub.Path(dpath)
>>> data_fpath = dpath / 'test.pkl'
>>> data = {'demo': 'data'}
>>> with open(str(data_fpath), 'wb') as file:
>>>     pickle.dump(data, file)
>>> # Write data
>>> import zipfile
>>> zip_fpath = dpath / 'test_zip.archive'
>>> stl_w_zfile = zipfile.ZipFile(os.fspath(zip_fpath), mode='w')
>>> stl_w_zfile.write(os.fspath(data_fpath), os.fspath(data_fpath.relative_
↪to(dpath)))
>>> stl_w_zfile.close()
>>> stl_r_zfile = zipfile.ZipFile(os.fspath(zip_fpath), mode='r')
>>> stl_r_zfile.namelist()
>>> stl_r_zfile.close()
>>> # Test zopen
>>> self = zopen(zip_fpath / 'test.pkl', mode='rb', ext='.archive')
>>> print(self._split_archive())
>>> print(self.namelist())
>>> self.close()
>>> self = zopen(zip_fpath / 'test.pkl', mode='rb', ext='.archive')
>>> recon1 = pickle.loads(self.read())
>>> self.close()
>>> self = zopen(zip_fpath / 'test.pkl', mode='rb', ext='.archive')
>>> recon2 = pickle.load(self)
>>> self.close()
>>> assert recon1 == recon2
>>> assert recon1 is not recon2
```

Example

```
>>> # Test we can load json data from a zipfile
>>> from ubelt.util_zip import *  # NOQA
>>> import ubelt as ub
>>> import json
>>> import zipfile
>>> dpath = ub.ensure_app_cache_dir('ubelt/tests/util_zip')
>>> infopath = join(dpath, 'info.json')
>>> ub.writeto(infopath, '{"x": "1"}')
>>> zippath = join(dpath, 'infozip.zip')
>>> internal = 'folder/info.json'
>>> with zipfile.ZipFile(zippath, 'w') as myzip:
>>>     myzip.write(infopath, internal)
>>> fpath = zippath + '/' + internal
>>> # Test context manager
```

(continues on next page)

(continued from previous page)

```
>>> with zopen(fpath, 'r') as self:
>>>     info2 = json.load(self)
>>>     assert info2['x'] == '1'
>>> # Test outside of context manager
>>> self = zopen(fpath, 'r')
>>> print(self._split_archive())
>>> info2 = json.load(self)
>>> assert info2['x'] == '1'
>>> # Test nice repr (with zfile)
>>> print('self = {!r}'.format(self))
>>> self.close()
```

Example

```
>>> # Coverage tests --- move to unit-test
>>> from ubelt.util_zip import *  # NOQA
>>> import ubelt as ub
>>> import json
>>> import zipfile
>>> dpath = ub.ensure_app_cache_dir('ubelt/tests/util_zip')
>>> textpath = join(dpath, 'seekable_test.txt')
>>> text = chr(10).join(['line{}'.format(i) for i in range(10)])
>>> ub.writeto(textpath, text)
>>> zippath = join(dpath, 'seekable_test.zip')
>>> internal = 'folder/seekable_test.txt'
>>> with zipfile.ZipFile(zippath, 'w') as myzip:
>>>     myzip.write(textpath, internal)
>>> ub.delete(textpath)
>>> fpath = zippath + '/' + internal
>>> # Test seekable
>>> self_seekable = zopen(fpath, 'r', seekable=True)
>>> assert self_seekable.seekable()
>>> self_seekable.seek(8)
>>> assert self_seekable.readline() == 'ne1' + chr(10)
>>> assert self_seekable.readline() == 'line2' + chr(10)
>>> self_seekable.seek(8)
>>> assert self_seekable.readline() == 'ne1' + chr(10)
>>> assert self_seekable.readline() == 'line2' + chr(10)
>>> # Test non-seekable?
>>> # Sometimes non-seekable files are still seekable
>>> maybe_seekable = zopen(fpath, 'r', seekable=False)
>>> if maybe_seekable.seekable():
>>>     maybe_seekable.seek(8)
>>>     assert maybe_seekable.readline() == 'ne1' + chr(10)
>>>     assert maybe_seekable.readline() == 'line2' + chr(10)
>>>     maybe_seekable.seek(8)
>>>     assert maybe_seekable.readline() == 'ne1' + chr(10)
>>>     assert maybe_seekable.readline() == 'line2' + chr(10)
```

Example

```
>>> # More coverage tests --- move to unit-test
>>> from ubelt.util_zip import *  # NOQA
>>> import ubelt as ub
>>> import pytest
>>> dpath = ub.ensure_app_cache_dir('ubelt/tests/util_zip')
>>> with pytest.raises(OSError):
>>>     self = zopen('', 'r')
>>> # Test open non-zip exsting file
>>> existing_fpath = join(dpath, 'exists.json')
>>> ub.writeto(existing_fpath, '{"x": "1"}')
>>> self = zopen(existing_fpath, 'r')
>>> assert self.read() == '{"x": "1"}'
>>> # Test dir
>>> dir(self)
>>> # Test nice
>>> print(self)
>>> print('self = {!r}'.format(self))
>>> self.close()
>>> # Test open non-zip non-existing file
>>> nonexisting_fpath = join(dpath, 'does-not-exist.txt')
>>> ub.delete(nonexisting_fpath)
>>> with pytest.raises(OSError):
>>>     self = zopen(nonexisting_fpath, 'r')
>>> with pytest.raises(NotImplementedError):
>>>     self = zopen(nonexisting_fpath, 'w')
>>> # Test nice-repr
>>> self = zopen(existing_fpath, 'r')
>>> print('self = {!r}'.format(self))
>>> # pathological
>>> self = zopen(existing_fpath, 'r')
>>> self._handle = None
>>> dir(self)
```

property zfile

Access the underlying archive file

namelist()

Lists the contents of this zipfile

ubelt.util_zip.**split_archive**(*fpath*, *ext='.zip'*)

If fpath specifies a file inside a zipfile, it breaks it into two parts the path to the zipfile and the internal path in the zipfile.

Example

```
>>> split_archive('/a/b/foo.txt')
>>> split_archive('/a/b/foo.zip/bar.txt')
>>> split_archive('/a/b/foo.zip/baz/biz.zip/bar.py')
>>> split_archive('archive.zip')
>>> import ubelt as ub
>>> split_archive(ub.Path('/a/b/foo.zip/baz/biz.zip/bar.py'))
>>> split_archive('/a/b/foo.zip/baz.pt/bar.zip/bar.zip', '.pt')
```

Todo: Fix got/want for win32

(None, None) ('/a/b/foo.zip', 'bar.txt') ('/a/b/foo.zip/baz/biz.zip', 'bar.py') ('archive.zip', None) ('/a/b/foo.zip/baz/biz.zip', 'bar.py') ('/a/b/foo.zip/baz.pt', 'bar.zip/bar.zip')

1.27.1.1.2 Module contents

UBelt is a "utility belt" of commonly needed utility and helper functions. It is a currated collection of top-level utilities with functionality that falls into a mixture of categories.

The source code is available at https://github.com/Erotemic/ubelt. We also have Jupyter notebook demos.

The ubelt API is organized by submodules containing related functionality. Each submodule contains top level overview documentation, and each function contains a docstring with at least one example.

NOTE: The README on github contains information and examples complementary to these docs.

class ubelt.**AutoDict**

Bases: `dict`

An infinitely nested default dict of dicts.

Implementation of Perl's autovivification feature.

SeeAlso: *AutoOrderedDict* - the ordered version

References

Example

```
>>> import ubelt as ub
>>> auto = ub.AutoDict()
>>> auto[0][10][100] = None
>>> assert str(auto) == '{0: {10: {100: None}}}'
```

to_dict()

Recursively casts a AutoDict into a regular dictionary. All nested AutoDict values are also converted.

Returns a copy of this dict without autovivification

Return type dict

Example

```
>>> from ubelt.util_dict import AutoDict
>>> auto = AutoDict()
>>> auto[1] = 1
>>> auto['n1'] = AutoDict()
>>> static = auto.to_dict()
>>> assert not isinstance(static, AutoDict)
>>> assert not isinstance(static['n1'], AutoDict)
```

class ubelt.**AutoOrderedDict**

Bases: `collections.OrderedDict`, *`ubelt.util_dict.AutoDict`*

An infinitely nested default dict of dicts that maintains the ordering of items.

SeeAlso: *`AutoDict`* - the unordered version of this class

Example

```
>>> import ubelt as ub
>>> auto = ub.AutoOrderedDict()
>>> auto[0][3] = 3
>>> auto[0][2] = 2
>>> auto[0][1] = 1
>>> assert list(auto[0].values()) == [3, 2, 1]
```

class ubelt.**CacheStamp**(*fname*, *dpath*, *cfgstr=None*, *product=None*, *hasher='sha1'*, *verbose=None*, *enabled=True*, *depends=None*, *meta=None*, *hash_prefix=None*, *expires=None*, *ext='.pkl'*)

Bases: `object`

Quickly determine if a file-producing computation has been done.

Check if the computation needs to be redone by calling `expired`. If the stamp is not expired, the user can expect that the results exist and could be loaded. If the stamp is expired, the computation should be redone. After the result is updated, the calls `renew`, which writes a "stamp" file to disk that marks that the procedure has been done.

There are several ways to control how a stamp expires. At a bare minimum, removing the stamp file will force expiration. However, in this circumstance CacheStamp only knows that something has been done, but it doesn't have any information about what was done, so in general this is not sufficient.

To achieve more robust expiration behavior, the user should specify the `product` argument, which is a list of file paths that are expected to exist whenever the stamp is renewed. When this is specified the CacheStamp will expire if any of these products are deleted, their size changes, their modified timestamp changes, or their hash (i.e. checksum) changes. Note that by setting `hasher=None`, running and verifying checksums can be disabled.

If the user knows what the hash of the file should be this can be specified to prevent renewal of the stamp unless these match the files on disk. This can be useful for security purposes.

The stamp can also be set to expire at a specified time or after a specified duration using the `expires` argument.

Parameters

- **fname** (*str*) – Name of the stamp file
- **dpath** (*str | PathLike | None*) – Where to store the cached stamp file

- **product** (*str | PathLike | Sequence[str | PathLike] | None*) – Path or paths that we expect the computation to produce. If specified the hash of the paths are stored.
- **hasher** (*str, default='sha1'*) – The type of hasher used to compute the file hash of product. If None, then we assume the file has not been corrupted or changed if the mtime and size are the same. Defaults to sha1.
- **verbose** (*bool, default=None*) – Passed to internal `ubelt.Cacher` object
- **enabled** (*bool, default=True*) – if False, expired always returns True
- **depends** (*str | List[str] | None*) – Indicate dependencies of this cache. If the dependencies change, then the cache is recomputed. New to CacheStamp in version 0.9.2, replaces `cfgstr`.
- **meta** (*object | None*) – Metadata that is also saved with the `cfgstr`. This can be useful to indicate how the `cfgstr` was constructed. New to CacheStamp in version 0.9.2. Note: this is a candidate for deprecation.
- **expires** (*str | int | datetime.datetime | datetime.timedelta | None*) – If specified, sets an expiration date for the certificate. This can be an absolute datetime or a timedelta offset. If specified as an int, this is interpreted as a time delta in seconds. If specified as a str, this is interpreted as an absolute timestamp. Time delta offsets are coerced to absolute times at "renew" time.
- **hash_prefix** (*None | str | List[str]*) – If specified, we verify that these match the hash(s) of the product(s) in the stamp certificate.
- **ext** (*str, default='.pkl'*) – File extension for the cache format. Can be `'.pkl'` or `'.json'`.
- **cfgstr** (*str | None*) – DEPRECATED in favor or depends.

Example

```
>>> import ubelt as ub
>>> # Stamp the computation of expensive-to-compute.txt
>>> dpath = ub.Path.appdir('ubelt/tests/cache-stamp')
>>> dpath.delete().ensuredir()
>>> product = dpath / 'expensive-to-compute.txt'
>>> self = ub.CacheStamp('somedata', depends='someconfig', dpath=dpath,
>>>                      product=product, hasher='sha256')
>>> self.clear()
>>> print(f'self.fpath={self.fpath}')
>>> if self.expired():
>>>     product.write_text('very expensive')
>>>     self.renew()
>>> assert not self.expired()
>>> # corrupting the output will cause the stamp to expire
>>> product.write_text('very corrupted')
>>> assert self.expired()
```

property fpath

clear()

Delete the stamp (the products are untouched)

expired(*cfgstr=None*, *product=None*)

Check to see if a previously existing stamp is still valid, if the expected result of that computation still exists, and if all other expiration criteria are met.

Parameters

- **cfgstr** (*str* | *None*) – overrides the instance-level cfgstr. DEPRECATED do not use.
- **product** (*str* | *PathLike* | *Sequence[str* | *PathLike]* | *None*) – override the default product if specified. DEPRECATED do not use.

Returns True(-thy) if the stamp is invalid, expired, or does not exist. When the stamp is expired, the reason for expiration is returned as a string. If the stamp is still valid, False is returned.

Return type bool | str

Example

```
>>> import ubelt as ub
>>> import time
>>> import os
>>> # Stamp the computation of expensive-to-compute.txt
>>> dpath = ub.Path.appdir('ubelt/tests/cache-stamp-expired')
>>> dpath.delete().ensuredir()
>>> products = [
>>>     dpath / 'product1.txt',
>>>     dpath / 'product2.txt',
>>> ]
>>> self = ub.CacheStamp('myname', depends='myconfig', dpath=dpath,
>>>                      product=products, hasher='sha256',
>>>                      expires=0)
>>> if self.expired():
>>>     for fpath in products:
>>>         fpath.write_text(fpath.name)
>>>     self.renew()
>>> fpath = products[0]
>>> # Because we set the expiration delta to 0, we should already be expired
>>> assert self.expired() == 'expired_cert'
>>> # Disable the expiration date, renew and we should be ok
>>> self.expires = None
>>> self.renew()
>>> assert not self.expired()
>>> # Modify the mtime to cause expiration
>>> orig_atime = fpath.stat().st_atime
>>> orig_mtime = fpath.stat().st_mtime
>>> os.utime(fpath, (orig_atime, orig_mtime + 200))
>>> assert self.expired() == 'mtime_diff'
>>> self.renew()
>>> assert not self.expired()
>>> # rewriting the file will cause the size constraint to fail
>>> # even if we hack the mtime to be the same
>>> orig_atime = fpath.stat().st_atime
>>> orig_mtime = fpath.stat().st_mtime
>>> fpath.write_text('corrupted')
>>> os.utime(fpath, (orig_atime, orig_mtime))
>>> assert self.expired() == 'size_diff'
>>> self.renew()
>>> assert not self.expired()
```

(continues on next page)

(continued from previous page)

```
>>> # Force a situation where the hash is the only thing
>>> # that saves us, write a different file with the same
>>> # size and mtime.
>>> orig_atime = fpath.stat().st_atime
>>> orig_mtime = fpath.stat().st_mtime
>>> fpath.write_text('corrApted')
>>> os.utime(fpath, (orig_atime, orig_mtime))
>>> assert self.expired() == 'hash_diff'
>>> # Test what a wrong hash prefix causes expiration
>>> certificate = self.renew()
>>> self.hash_prefix = certificate['hash']
>>> self.expired()
>>> self.hash_prefix = ['bad', 'hashes']
>>> self.expired()
>>> # A bad hash will not allow us to renew
>>> import pytest
>>> with pytest.raises(RuntimeError):
...     self.renew()
```

renew(*cfgstr=None*, *product=None*)

Recertify that the product has been recomputed by writing a new certificate to disk.

Returns certificate information

Return type dict

class ubelt.**Cacher**(*fname*, *depends=None*, *dpath=None*, *appname='ubelt'*, *ext='.pkl'*, *meta=None*, *verbose=None*, *enabled=True*, *log=None*, *hasher='sha1'*, *protocol=- 1*, *cfgstr=None*, *backend='auto'*)

Bases: object

Saves data to disk and reloads it based on specified dependencies.

Cacher uses pickle to save/load data to/from disk. Dependencies of the cached process can be specified, which ensures the cached data is recomputed if the dependencies change. If the location of the cache is not specified, it will default to the system user's cache directory.

Parameters

- **fname** (*str*) – A file name. This is the prefix that will be used by the cache. It will always be used as-is.
- **depends** (*str | List[str] | None*) – Indicate dependencies of this cache. If the dependencies change, then the cache is recomputed. New in version 0.8.9, replaces `cfgstr`.
- **dpath** (*str | PathLike | None*) – Specifies where to save the cache. If unspecified, Cacher defaults to an application cache dir as given by appname. See `ub.get_app_cache_dir()` for more details.
- **appname** (*str, default='ubelt'*) – Application name Specifies a folder in the application cache directory where to cache the data if `dpath` is not specified.
- **ext** (*str, default='.pkl'*) – File extension for the cache format. Can be `'.pkl'` or `'.json'`.
- **meta** (*object | None*) – Metadata that is also saved with the `cfgstr`. This can be useful to indicate how the `cfgstr` was constructed. Note: this is a candidate for deprecation.
- **verbose** (*int, default=1*) – Level of verbosity. Can be 1, 2 or 3.

- **enabled** (*bool, default=True*) – If set to False, then the load and save methods will do nothing.
- **log** (*Callable[[str], Any]*) – Overloads the print function. Useful for sending output to loggers (e.g. logging.info, tqdm.tqdm.write, ...)
- **hasher** (*str, default='sha1'*) – Type of hashing algorithm to use if `cfgstr` needs to be condensed to less than 49 characters.
- **protocol** (*int, default=-1*) – Protocol version used by pickle. Defaults to the -1 which is the latest protocol.
- **backend** (*str*) – Set to either `'pickle'` or `'json'` to force backend. Defaults to auto which chooses one based on the extension.
- **cfgstr** (*str | None*) – Deprecated in favor of `depends`.

Example

```
>>> import ubelt as ub
>>> depends = 'repr-of-params-that-uniquely-determine-the-process'
>>> # Create a cacher and try loading the data
>>> cacher = ub.Cacher('demo_process', depends, verbose=4)
>>> cacher.clear()
>>> print(f'cacher.fpath={cacher.fpath}')
>>> data = cacher.tryload()
>>> if data is None:
>>>     # Put expensive functions in if block when cacher misses
>>>     myvar1 = 'result of expensive process'
>>>     myvar2 = 'another result'
>>>     # Tell the cacher to write at the end of the if block
>>>     # It is idomatic to put results in an object named data
>>>     data = myvar1, myvar2
>>>     cacher.save(data)
>>> # Last part of the Cacher pattern is to unpack the data object
>>> myvar1, myvar2 = data
>>> #
>>> # If we know the data exists, we can also simply call load
>>> data = cacher.tryload()
```

Example

```
>>> # The previous example can be shorted if only a single value
>>> from ubelt.util_cache import Cacher
>>> depends = 'repr-of-params-that-uniquely-determine-the-process'
>>> # Create a cacher and try loading the data
>>> cacher = Cacher('demo_process', depends)
>>> myvar = cacher.tryload()
>>> if myvar is None:
>>>     myvar = ('result of expensive process', 'another result')
>>>     cacher.save(myvar)
>>> assert cacher.exists(), 'should now exist'
```

VERBOSE = 1

FORCE_DISABLE = False

property fpath

get_fpath(*cfgstr=None*)

Reports the filepath that the cacher will use.

It will attempt to use '{fname}_{cfgstr}{ext}' unless that is too long. Then cfgstr will be hashed.

Parameters **cfgstr** (*str | None*) – overrides the instance-level cfgstr

Returns str | PathLike

Example

```
>>> # xdoctest: +REQUIRES(module:pytest)
>>> from ubelt.util_cache import Cacher
>>> import pytest
>>> with pytest.warns(UserWarning):
>>>     cacher = Cacher('test_cacher1')
>>>     cacher.get_fpath()
>>> self = Cacher('test_cacher2', depends='cfg1')
>>> self.get_fpath()
>>> self = Cacher('test_cacher3', depends='cfg1' * 32)
>>> self.get_fpath()
```

exists(*cfgstr=None*)

Check to see if the cache exists

Parameters **cfgstr** (*str | None*) – overrides the instance-level cfgstr

Returns bool

existing_versions()

Returns data with different cfgstr values that were previously computed with this cacher.

Yields *str* – paths to cached files corresponding to this cacher

Example

```
>>> from ubelt.util_cache import Cacher
>>> # Ensure that some data exists
>>> known_fpaths = set()
>>> import ubelt as ub
>>> dpath = ub.ensure_app_cache_dir('ubelt',
>>>                                 'test-existing-versions')
>>> ub.delete(dpath)  # start fresh
>>> cacher = Cacher('versioned_data_v2', depends='1', dpath=dpath)
>>> cacher.ensure(lambda: 'data1')
>>> known_fpaths.add(cacher.get_fpath())
>>> cacher = Cacher('versioned_data_v2', depends='2', dpath=dpath)
>>> cacher.ensure(lambda: 'data2')
>>> known_fpaths.add(cacher.get_fpath())
>>> # List previously computed configs for this type
```

(continues on next page)

(continued from previous page)

```
>>> from os.path import basename
>>> cacher = Cacher('versioned_data_v2', depends='2', dpath=dpath)
>>> exist_fpaths = set(cacher.existing_versions())
>>> exist_fnames = list(map(basename, exist_fpaths))
>>> print('exist_fnames = {!r}'.format(exist_fnames))
>>> assert exist_fpaths.issubset(known_fpaths)
```

clear(*cfgstr=None*)

Removes the saved cache and metadata from disk

Parameters **cfgstr** (*str | None*) – overrides the instance-level cfgstr

tryload(*cfgstr=None*, *on_error='raise'*)

Like load, but returns None if the load fails due to a cache miss.

Parameters

- **cfgstr** (*str | None*) – overrides the instance-level cfgstr
- **on_error** (*str, default='raise'*) – How to handle non-io errors errors. Either 'raise', which re-raises the exception, or 'clear' which deletes the cache and returns None.

Returns the cached data if it exists, otherwise returns None

Return type None | object

load(*cfgstr=None*)

Load the data cached and raise an error if something goes wrong.

Parameters **cfgstr** (*str | None*) – overrides the instance-level cfgstr

Returns the cached data

Return type object

Raises **IOError - if the data is unable to be loaded. This could be due to** – a cache miss or because the cache is disabled.

Example

```
>>> from ubelt.util_cache import *  # NOQA
>>> # Setting the cacher as enabled=False turns it off
>>> cacher = Cacher('test_disabled_load', '', enabled=True)
>>> cacher.save('data')
>>> assert cacher.load() == 'data'
>>> cacher.enabled = False
>>> assert cacher.tryload() is None
```

save(*data*, *cfgstr=None*)

Writes data to path specified by `self.fpath`.

Metadata containing information about the cache will also be appended to an adjacent file with the *.meta* suffix.

Parameters

- **data** (*object*) – arbitrary pickleable object to be cached
- **cfgstr** (*str | None*) – overrides the instance-level cfgstr

Example

```
>>> from ubelt.util_cache import *  # NOQA
>>> # Normal functioning
>>> depends = 'long-cfg' * 32
>>> cacher = Cacher('test_enabled_save', depends=depends)
>>> cacher.save('data')
>>> assert exists(cacher.get_fpath()), 'should be enabled'
>>> assert exists(cacher.get_fpath() + '.meta'), 'missing metadata'
>>> # Setting the cacher as enabled=False turns it off
>>> cacher2 = Cacher('test_disabled_save', 'params', enabled=False)
>>> cacher2.save('data')
>>> assert not exists(cacher2.get_fpath()), 'should be disabled'
```

ensure(*func*, **args*, ***kwargs*)

Wraps around a function. A cfgstr must be stored in the base cacher.

Parameters

- **func** (*Callable*) – function that will compute data on cache miss
- ***args** – passed to func
- ****kwargs** – passed to func

Example

```
>>> from ubelt.util_cache import *  # NOQA
>>> def func():
>>>     return 'expensive result'
>>> fname = 'test_cacher_ensure'
>>> depends = 'func params'
>>> cacher = Cacher(fname, depends=depends)
>>> cacher.clear()
>>> data1 = cacher.ensure(func)
>>> data2 = cacher.ensure(func)
>>> assert data1 == 'expensive result'
>>> assert data1 == data2
>>> cacher.clear()
```

class ubelt.**CaptureStdout**(*suppress=True*, *enabled=True*)

Bases: *ubelt.util_stream.CaptureStream*

Context manager that captures stdout and stores it in an internal stream

Parameters

- **suppress** (*bool, default=True*) – if True, stdout is not printed while captured
- **enabled** (*bool, default=True*) – does nothing if this is False
- ****kwargs** – used for backwards compatibility with misspelled deprecated params.

Example

```
>>> import ubelt as ub
>>> self = ub.CaptureStdout(suppress=True)
>>> print('dont capture the table flip (°° ')
>>> with self:
...     text = 'capture the heart '
...     print(text)
>>> print('dont capture look of disapproval _')
>>> assert isinstance(self.text, str)
>>> assert self.text == text + '\n', 'failed capture text'
```

Example

```
>>> import ubelt as ub
>>> self = ub.CaptureStdout(suppress=False)
>>> with self:
...     print('I am captured and printed in stdout')
>>> assert self.text.strip() == 'I am captured and printed in stdout'
```

Example

```
>>> import ubelt as ub
>>> self = ub.CaptureStdout(suppress=True, enabled=False)
>>> with self:
...     print('dont capture')
>>> assert self.text is None
```

log_part()

Log what has been captured so far

start()

stop()

Example

```
>>> import ubelt as ub
>>> ub.CaptureStdout(enabled=False).stop()
>>> ub.CaptureStdout(enabled=True).stop()
```

close()

class ubelt.**CaptureStream**

Bases: object

Generic class for capturing streaming output from stdout or stderr

class ubelt.**DownloadManager**(*download_root=None*, *mode='thread'*, *max_workers=None*, *cache=True*)

Bases: `object`

Simple implementation of the download manager

Variables

- **download_root** (*str | PathLike*) – default download location
- **jobs** (*List[concurrent.futures.Future]*) – list of jobs

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import ubelt as ub
>>> # Download a file with a known hash
>>> manager = ub.DownloadManager()
>>> job = manager.submit(
>>>     'http://i.imgur.com/rqwaDag.png',
>>>     hash_prefix=
↪'31a129618c87dd667103e7154182e3c39a605eefe90f84f2283f3c87efee8e40'
>>> )
>>> fpath = job.result()
>>> print('fpath = {!r}'.format(fpath))
```

Example

```
>>> # Does not require network
>>> import ubelt as ub
>>> manager = ub.DownloadManager()
>>> for i in range(100):
...     job = manager.submit('localhost/might-not-exist-i-{}'.format(i))
>>> file_paths = []
>>> for job in manager.as_completed(prog=True):
...     try:
...         fpath = job.result()
...         file_paths += [fpath]
...     except Exception:
...         pass
>>> print('file_paths = {!r}'.format(file_paths))
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import pytest
>>> import ubelt as ub
>>> manager = ub.DownloadManager()
>>> item1 = {
>>>     'url': 'https://data.kitware.com/api/v1/item/5b4039308d777f2e6225994c/
↪download',
>>>     'dst': 'forgot_what_the_name_really_is',
```

(continues on next page)

(continued from previous page)

```
>>>     'hash_prefix': 'c98a46cb31205cf',
>>>     'hasher': 'sha512',
>>> }
>>> item2 = {
>>>     'url': 'http://i.imgur.com/rqwaDag.png',
>>>     'hash_prefix': 'f79ea24571da6ddd2ba12e3d57b515249ecb8a35',
>>>     'hasher': 'sha1',
>>> }
>>> item1 = item2  # hack around SSL error
>>> manager.submit(**item1)
>>> manager.submit(**item2)
>>> for job in manager.as_completed(prog=True, verbose=3):
>>>     fpath = job.result()
>>>     print('fpath = {!r}'.format(fpath))
```

submit(*url*, *dst=None*, *hash_prefix=None*, *hasher='sha256'*)

Add a job to the download Queue

Parameters

- **url** (*str | PathLike*) – pointer to the data to download
- **dst** (*str | None*) – The relative or absolute path to download to. If unspecified, the destination name is derived from the url.
- **hash_prefix** (*str | None*) – If specified, verifies that the hash of the downloaded file starts with this.
- **hasher** (*str, default='sha256'*) – hashing algorithm to use if hash_prefix is specified.

Returns a Future object that will point to the downloaded location.

Return type concurrent.futures.Future

as_completed(*prog=None*, *desc=None*, *verbose=1*)

Generate completed jobs as they become available

Example

```
>>> import pytest
>>> import ubelt as ub
>>> download_root = ub.ensure_app_config_dir('ubelt', 'dlman')
>>> manager = ub.DownloadManager(download_root=download_root,
>>>                              cache=False)
>>> for i in range(3):
>>>     manager.submit('localhost')
>>> results = list(manager)
>>> print('results = {!r}'.format(results))
>>> manager.shutdown()
```

shutdown()

Cancel all jobs and close all connections.

class ubelt.**Executor**(*mode='thread', max_workers=0*)

Bases: object

Wrapper around a specific executor.

Abstracts Serial, Thread, and Process Executor via arguments.

Parameters

- **mode** (*str, default='thread'*) – either thread, serial, or process
- **max_workers** (*int, default=0*) – number of workers. If 0, serial is forced.

Example

```
>>> import platform
>>> import sys
>>> # The process backend breaks pyp3 when using coverage
>>> if 'pypy' in platform.python_implementation().lower():
...     import pytest
...     pytest.skip('not testing process on pypy')
>>> if sys.platform.startswith('win32'):
...     import pytest
...     pytest.skip('not running this test on win32 for now')
>>> import ubelt as ub
>>> # Fork before threading!
>>> # https://pybay.com/site_media/slides/raymond2017-keynote/combo.html
>>> self1 = ub.Executor(mode='serial', max_workers=0)
>>> self1.__enter__()
>>> self2 = ub.Executor(mode='process', max_workers=2)
>>> self2.__enter__()
>>> self3 = ub.Executor(mode='thread', max_workers=2)
>>> self3.__enter__()
>>> jobs = []
>>> jobs.append(self1.submit(sum, [1, 2, 3]))
>>> jobs.append(self1.submit(sum, [1, 2, 3]))
>>> jobs.append(self2.submit(sum, [10, 20, 30]))
>>> jobs.append(self2.submit(sum, [10, 20, 30]))
>>> jobs.append(self3.submit(sum, [4, 5, 5]))
>>> jobs.append(self3.submit(sum, [4, 5, 5]))
>>> for job in jobs:
>>>     result = job.result()
>>>     print('result = {!r}'.format(result))
>>> self1.__exit__(None, None, None)
>>> self2.__exit__(None, None, None)
>>> self3.__exit__(None, None, None)
```

Example

```
>>> import ubelt as ub
>>> self1 = ub.Executor(mode='serial', max_workers=0)
>>> with self1:
>>>     jobs = []
>>>     for i in range(10):
>>>         jobs.append(self1.submit(sum, [i + 1, i]))
>>>     for job in jobs:
>>>         job.add_done_callback(lambda x: print('done callback got x = {}'.
↪format(x)))
>>>         result = job.result()
>>>         print('result = {!r}'.format(result))
```

submit(*func*, **args*, ***kw*)

Calls the submit function of the underlying backend.

Returns a future representing the job

Return type concurrent.futures.Future

shutdown()

Calls the shutdown function of the underlying backend.

map(*fn*, **iterables*, ***kwargs*)

Calls the map function of the underlying backend.

CommandLine

```
xdoctest -m /home/joncrall/code/ubelt/ubelt/util_futures.py Executor.map
```

Example

```
>>> import ubelt as ub
>>> import concurrent.futures
>>> import string
>>> with ub.Executor(mode='serial') as executor:
...     result_iter = executor.map(int, string.digits)
...     results = list(result_iter)
>>> print('results = {!r}'.format(results))
results = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> with ub.Executor(mode='thread', max_workers=2) as executor:
...     result_iter = executor.map(int, string.digits)
...     results = list(result_iter)
>>> # xdoctest: +IGNORE_WANT
>>> print('results = {!r}'.format(results))
results = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

class ubelt.**FormatterExtensions**

Bases: object

Helper class for managing non-builtin (e.g. numpy) format types.

This module (*ubelt.util_format*) maintains a global set of basic extensions, but it is also possible to create a locally scoped set of extensions and explicitly pass it to repr2. The following example demonstrates this.

Example

```
>>> import ubelt as ub
>>> class MyObject(object):
>>>     pass
>>> data = {'a': [1, 2.2222, MyObject()], 'b': MyObject()}
>>> # Create a custom set of extensions
>>> extensions = ub.FormatterExtensions()
>>> # Register a function to format your specific type
>>> @extensions.register(MyObject)
>>> def format_myobject(data, **kwargs):
>>>     return 'I can do anything here'
>>> # Repr2 will now respect the passed custom extensions
>>> # Note that the global extensions will still be respected
>>> # unless they are overloaded.
>>> print(ub.repr2(data, nl=-1, precision=1, extensions=extensions))
{
    'a': [1, 2.2, I can do anything here],
    'b': I can do anything here
}
>>> # Overload the formatter for float and int
>>> @extensions.register((float, int))
>>> def format_myobject(data, **kwargs):
>>>     return str((data + 10) // 2)
>>> print(ub.repr2(data, nl=-1, precision=1, extensions=extensions))
{
    'a': [5, 6.0, I can do anything here],
    'b': I can do anything here
}
```

register(*key*)

Registers a custom formatting function with ub.repr2

Parameters **key** (*Type | Tuple[Type] | str*) – indicator of the type

Returns decorator function

Return type Callable

lookup(*data*)

Returns an appropriate function to format `data` if one has been registered.

class ubelt.**IndexableWalker**(*data*, *dict_cls=(<class 'dict'>,)*, *list_cls=(<class 'list'>, <class 'tuple'>)*)

Bases: `collections.abc.Generator`

Traverses through a nested tree-liked indexable structure.

Generates a path and value to each node in the structure. The path is a list of indexes which if applied in order will reach the value.

The `__setitem__` method can be used to modify a nested value based on the path returned by the generator.

When generating values, you can use "send" to prevent traversal of a particular branch.

RelatedWork:

- **https://pypi.org/project/python-benedict/ - implements a dictionary** subclass with similar nested indexing abilities.

Example

```
>>> # Given Nested Data
>>> data = {
>>>     'foo': {'bar': 1},
>>>     'baz': [{'biz': 3}, {'buz': [4, 5, 6]}],
>>> }
>>> # Create an IndexableWalker
>>> walker = IndexableWalker(data)
>>> # We iterate over the data as if it was flat
>>> # ignore the <want> string due to order issues on older Pythons
>>> # xdoctest: +IGNORE_WANT
>>> for path, val in walker:
>>>     print(path)
['foo']
['baz']
['baz', 0]
['baz', 1]
['baz', 1, 'buz']
['baz', 1, 'buz', 0]
['baz', 1, 'buz', 1]
['baz', 1, 'buz', 2]
['baz', 0, 'biz']
['foo', 'bar']
>>> # We can use "paths" as keys to getitem into the walker
>>> path = ['baz', 1, 'buz', 2]
>>> val = walker[path]
>>> assert val == 6
>>> # We can use "paths" as keys to setitem into the walker
>>> assert data['baz'][1]['buz'][2] == 6
>>> walker[path] = 7
>>> assert data['baz'][1]['buz'][2] == 7
>>> # We can use "paths" as keys to delitem into the walker
>>> assert data['baz'][1]['buz'][1] == 5
>>> del walker[['baz', 1, 'buz', 1]]
>>> assert data['baz'][1]['buz'][1] == 7
```

Example

```
>>> # Create nested data
>>> # xdoctest: +REQUIRES(module:numpy)
>>> import numpy as np
>>> import ubelt as ub
>>> data = ub.ddict(lambda: int)
>>> data['foo'] = ub.ddict(lambda: int)
>>> data['bar'] = np.array([1, 2, 3])
```

(continues on next page)

(continued from previous page)

```
>>> data['foo']['a'] = 1
>>> data['foo']['b'] = np.array([1, 2, 3])
>>> data['foo']['c'] = [1, 2, 3]
>>> data['baz'] = 3
>>> print('data = {}'.format(ub.repr2(data, nl=True)))
>>> # We can walk through every node in the nested tree
>>> walker = IndexableWalker(data)
>>> for path, value in walker:
>>>     print('walk path = {}'.format(ub.repr2(path, nl=0)))
>>>     if path[-1] == 'c':
>>>         # Use send to prevent traversing this branch
>>>         got = walker.send(False)
>>>         # We can modify the value based on the returned path
>>>         walker[path] = 'changed the value of c'
>>> print('data = {}'.format(ub.repr2(data, nl=True)))
>>> assert data['foo']['c'] == 'changed the value of c'
```

Example

```
>>> # Test sending false for every data item
>>> # xdoctest: +REQUIRES(CPython)
>>> # xdoctest: +REQUIRES(module:numpy)
>>> import ubelt as ub
>>> import numpy as np
>>> data = {1: 1}
>>> walker = IndexableWalker(data)
>>> for path, value in walker:
>>>     print('walk path = {}'.format(ub.repr2(path, nl=0)))
>>>     walker.send(False)
>>> data = {}
>>> walker = IndexableWalker(data)
>>> for path, value in walker:
>>>     walker.send(False)
```

send(*arg*) → send 'arg' into generator,

return next yielded value or raise StopIteration.

throw(*typ*[, *val*[, *tb*]]) → raise exception in generator,

return next yielded value or raise StopIteration.

class ubelt.**JobPool**(*mode='thread'*, *max_workers=0*)

Bases: `object`

Abstracts away boilerplate of submitting and collecting jobs

This is a basic wrapper around `ubelt.util_futures.Executor` that simplifies the most basic case.

Example

```
>>> import ubelt as ub
>>> def worker(data):
>>>     return data + 1
>>> pool = ub.JobPool('thread', max_workers=16)
>>> for data in ub.ProgIter(range(10), desc='submit jobs'):
>>>     pool.submit(worker, data)
>>> final = []
>>> for job in pool.as_completed(desc='collect jobs'):
>>>     info = job.result()
>>>     final.append(info)
>>> print('final = {!r}'.format(final))
```

submit(*func*, **args*, ***kwargs*)

Submit a job managed by the pool

Parameters

- **func** (*Callable[..., Any]*) – A callable that will take as many arguments as there are passed iterables.
- ***args** – positional arguments to pass to the function
- ***kwargs** – keyword arguments to pass to the function

Returns a future representing the job

Return type concurrent.futures.Future

shutdown()

as_completed(*timeout=None*, *desc=None*, *progkw=None*)

Generates completed jobs in an arbitrary order

Parameters

- **timeout** (*float | None*) – Specify the the maximum number of seconds to wait for a job.
- **desc** (*str | None*) – if specified, reports progress with a `ubelt.progiter.ProgIter` object.
- **progkw** (*dict | None*) – extra keyword arguments to `ubelt.progiter.ProgIter`.

Yields *concurrent.futures.Future* – The completed future object containing the results of a job.

CommandLine

```
xdoctest -m ubelt.util_futures JobPool.as_completed
```

Example

```
>>> import ubelt as ub
>>> pool = ub.JobPool('thread', max_workers=8)
>>> text = ub.paragraph(
...     '''
...     UDP is a cool protocol, check out the wiki:
...
...     UDP-based Data Transfer Protocol (UDT), is a high-performance
...     data transfer protocol designed for transferring large
...     volumetric datasets over high-speed wide area networks. Such
...     settings are typically disadvantageous for the more common TCP
...     protocol.
...     ''')
>>> for word in text.split(' '):
...     pool.submit(print, word)
>>> for _ in pool.as_completed():
...     pass
>>> pool.shutdown()
```

join(***kwargs*)

Like **:method:`JobPool.as_completed`**, but executes the *result* method of each future and returns only after all processes are complete. This allows for lower-boilerplate prototyping.

Parameters ****kwargs** – passed to **:method:`JobPool.as_completed`**

Returns list of results

Return type List[Any]

Example

```
>>> import ubelt as ub
>>> # We just want to try replacing our simple iterative algorithm
>>> # with the embarassingly parallel version
>>> arglist = list(zip(range(1000), range(1000)))
>>> func = ub.identity
>>> #
>>> # Original version
>>> for args in arglist:
>>>     func(*args)
>>> #
>>> # Potentially parallel version
>>> jobs = ub.JobPool(max_workers=0)
>>> for args in arglist:
>>>     jobs.submit(func, *args)
>>> _ = jobs.join(desc='running')
```

class ubelt.**NiceRepr**

Bases: object

Inherit from this class and define `__nice__` to "nicely" print your objects.

Defines `__str__` and `__repr__` in terms of `__nice__` function Classes that inherit from *NiceRepr* should redefine `__nice__`. If the inheriting class has a `__len__`, method then the default `__nice__` method will

return its length.

Example

```
>>> import ubelt as ub
>>> class Foo(ub.NiceRepr):
...     def __nice__(self):
...         return 'info'
>>> foo = Foo()
>>> assert str(foo) == '<Foo(info)>'
>>> assert repr(foo).startswith('<Foo(info) at ')
```

Example

```
>>> import ubelt as ub
>>> class Bar(ub.NiceRepr):
...     pass
>>> bar = Bar()
>>> import pytest
>>> with pytest.warns(RuntimeWarning) as record:
>>>     assert 'object at' in str(bar)
>>>     assert 'object at' in repr(bar)
```

Example

```
>>> import ubelt as ub
>>> class Baz(ub.NiceRepr):
...     def __len__(self):
...         return 5
>>> baz = Baz()
>>> assert str(baz) == '<Baz(5)>'
```

Example

```
>>> import ubelt as ub
>>> # If your nice message has a bug, it shouldn't bring down the house
>>> class Foo(ub.NiceRepr):
...     def __nice__(self):
...         assert False
>>> foo = Foo()
>>> import pytest
>>> with pytest.warns(RuntimeWarning) as record:
>>>     print('foo = {!r}'.format(foo))
foo = <...Foo ...>
```

class ubelt.**OrderedSet**(*iterable=None*)

Bases: `collections.abc.MutableSet`, `collections.abc.Sequence`

An OrderedSet is a custom MutableSet that remembers its order, so that every entry has an index that can be looked up.

Example

```
>>> OrderedSet([1, 1, 2, 3, 2])
OrderedSet([1, 2, 3])
```

copy()

Return a shallow copy of this object.

Example

```
>>> this = OrderedSet([1, 2, 3])
>>> other = this.copy()
>>> this == other
True
>>> this is other
False
```

add(*key*)

Add *key* as an item to this OrderedSet, then return its index.

If *key* is already in the OrderedSet, return the index it already had.

Example

```
>>> oset = OrderedSet()
>>> oset.append(3)
0
>>> print(oset)
OrderedSet([3])
```

append(*key*)

Add *key* as an item to this OrderedSet, then return its index.

If *key* is already in the OrderedSet, return the index it already had.

Example

```
>>> oset = OrderedSet()
>>> oset.append(3)
0
>>> print(oset)
OrderedSet([3])
```

update(*sequence*)

Update the set with the given iterable sequence, then return the index of the last element inserted.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.update([3, 1, 5, 1, 4])
4
>>> print(oset)
OrderedSet([1, 2, 3, 5, 4])
```

index(*key*)

Get the index of a given entry, raising an IndexError if it's not present.

key can be an iterable of entries that is not a string, in which case this returns a list of indices.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.index(2)
1
```

get_loc(*key*)

Get the index of a given entry, raising an IndexError if it's not present.

key can be an iterable of entries that is not a string, in which case this returns a list of indices.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.index(2)
1
```

get_indexer(*key*)

Get the index of a given entry, raising an IndexError if it's not present.

key can be an iterable of entries that is not a string, in which case this returns a list of indices.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.index(2)
1
```

pop()

Remove and return the last element from the set.

Raises KeyError if the set is empty.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.pop()
3
```

discard(*key*)

Remove an element. Do not raise an exception if absent.

The MutableSet mixin uses this to implement the .remove() method, which *does* raise an error when asked to remove a non-existent item.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.discard(2)
>>> print(oset)
OrderedSet([1, 3])
>>> oset.discard(2)
>>> print(oset)
OrderedSet([1, 3])
```

clear()

Remove all items from this OrderedSet.

union(**sets*)

Combines all unique items. Each items order is defined by its first appearance.

Example

```
>>> oset = OrderedSet.union(OrderedSet([3, 1, 4, 1, 5]), [1, 3], [2, 0])
>>> print(oset)
OrderedSet([3, 1, 4, 5, 2, 0])
>>> oset.union([8, 9])
OrderedSet([3, 1, 4, 5, 2, 0, 8, 9])
>>> oset | {10}
OrderedSet([3, 1, 4, 5, 2, 0, 10])
```

intersection(**sets*)

Returns elements in common between all sets. Order is defined only by the first set.

Example

```
>>> oset = OrderedSet.intersection(OrderedSet([0, 1, 2, 3]), [1, 2, 3])
>>> print(oset)
OrderedSet([1, 2, 3])
>>> oset.intersection([2, 4, 5], [1, 2, 3, 4])
OrderedSet([2])
>>> oset.intersection()
OrderedSet([1, 2, 3])
```

difference(**sets*)

Returns all elements that are in this set but not the others.

Example

```
>>> OrderedSet([1, 2, 3]).difference(OrderedSet([2]))
OrderedSet([1, 3])
>>> OrderedSet([1, 2, 3]).difference(OrderedSet([2]), OrderedSet([3]))
OrderedSet([1])
>>> OrderedSet([1, 2, 3]) - OrderedSet([2])
OrderedSet([1, 3])
>>> OrderedSet([1, 2, 3]).difference()
OrderedSet([1, 2, 3])
```

issubset(*other*)

Report whether another set contains this set.

Example

```
>>> OrderedSet([1, 2, 3]).issubset({1, 2})
False
>>> OrderedSet([1, 2, 3]).issubset({1, 2, 3, 4})
True
>>> OrderedSet([1, 2, 3]).issubset({1, 4, 3, 5})
False
```

issuperset(*other*)

Report whether this set contains another set.

Example

```
>>> OrderedSet([1, 2]).issuperset([1, 2, 3])
False
>>> OrderedSet([1, 2, 3, 4]).issuperset({1, 2, 3})
True
>>> OrderedSet([1, 4, 3, 5]).issuperset({1, 2, 3})
False
```

symmetric_difference(*other*)

Return the symmetric difference of two OrderedSets as a new set. That is, the new set will contain all elements that are in exactly one of the sets.

Their order will be preserved, with elements from *self* preceding elements from *other*.

Example

```
>>> this = OrderedSet([1, 4, 3, 5, 7])
>>> other = OrderedSet([9, 7, 1, 3, 2])
>>> this.symmetric_difference(other)
OrderedSet([4, 5, 9, 2])
```

difference_update(**sets*)

Update this OrderedSet to remove items from one or more other sets.

Example

```
>>> this = OrderedSet([1, 2, 3])
>>> this.difference_update(OrderedSet([2, 4]))
>>> print(this)
OrderedSet([1, 3])
```

```
>>> this = OrderedSet([1, 2, 3, 4, 5])
>>> this.difference_update(OrderedSet([2, 4]), OrderedSet([1, 4, 6]))
>>> print(this)
OrderedSet([3, 5])
```

intersection_update(*other*)

Update this OrderedSet to keep only items in another set, preserving their order in this set.

Example

```
>>> this = OrderedSet([1, 4, 3, 5, 7])
>>> other = OrderedSet([9, 7, 1, 3, 2])
>>> this.intersection_update(other)
>>> print(this)
OrderedSet([1, 3, 7])
```

symmetric_difference_update(*other*)

Update this OrderedSet to remove items from another set, then add items from the other set that were not present in this set.

Example

```
>>> this = OrderedSet([1, 4, 3, 5, 7])
>>> other = OrderedSet([9, 7, 1, 3, 2])
>>> this.symmetric_difference_update(other)
>>> print(this)
OrderedSet([4, 5, 9, 2])
```

class ubelt.**Path**(**args*, ***kwargs*)

Bases: `pathlib.PosixPath`

An extension of `pathlib.Path` with extra convenience methods

Note:

New methods are:

- augment
- ensuredir
- expand
- expandvars
- ls
- shrinkuser
- walk

New classmethods are:

- appdir

Modified methods are:

- touch

Example

```
>>> # Ubelt extends pathlib functionality
>>> import ubelt as ub
>>> dpath = ub.Path('~/.cache/ubelt/demo_path').expand().ensuredir()
>>> fpath = dpath / 'text_file.txt'
>>> aug_fpath = fpath.augment(suffix='.aux', ext='.jpg').touch()
>>> aug_dpath = dpath.augment('demo_path2')
>>> assert aug_fpath.read_text() == ''
>>> fpath.write_text('text data')
>>> assert aug_fpath.exists()
>>> assert not aug_fpath.delete().exists()
>>> assert dpath.exists()
>>> assert not dpath.delete().exists()
>>> print(f'{str(fpath.shrinkuser()).replace(os.path.sep, "/")}')
>>> print(f'{str(dpath.shrinkuser()).replace(os.path.sep, "/")}')
>>> print(f'{str(aug_fpath.shrinkuser()).replace(os.path.sep, "/")}')
>>> print(f'{str(aug_dpath.shrinkuser()).replace(os.path.sep, "/")}')
~/.cache/ubelt/demo_path/text_file.txt
~/.cache/ubelt/demo_path
~/.cache/ubelt/demo_path/text_file.aux.jpg
~/.cache/ubelt/demo_pathdemo_path2
```

classmethod appdir(*appname*, **args*, *type='cache'*)

Returns an operating system appropriate writable directory for an application to be used for cache, configs, or data.

Parameters

- **appname** (*str*) – the name of the application
- ***args[str]** – optional subdirs

- **type** (*str*) – can be 'cache', 'config', or 'data'.

Returns a new path object

Return type *Path*

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> import ubelt as ub
>>> print(ub.Path.appdir('ubelt', type='cache').shrinkuser())
>>> print(ub.Path.appdir('ubelt', type='config').shrinkuser())
>>> print(ub.Path.appdir('ubelt', type='data').shrinkuser())
~/.cache/ubelt
~/.config/ubelt
~/.local/share/ubelt
>>> import pytest
>>> with pytest.raises(KeyError):
>>>     ub.Path.appdir('ubelt', type='other')
```

augment(*suffix=''*, *prefix=''*, *ext=None*, *stem=None*, *dpath=None*, *tail=''*, *relative=None*, *multidot=False*)

Create a new path with a different extension, basename, directory, prefix, and/or suffix.

See *augpath()* for more details.

Parameters

- **suffix** (*str*) – Text placed between the stem and extension. Default to ''.
- **prefix** (*str*) – Text placed in front of the stem. Defaults to ''.
- **ext** (*str | None*) – If specified, replaces the extension
- **stem** (*str | None*) – If specified, replaces the stem (i.e. basename without extension). Note: named base in *augpath()*.
- **dpath** (*str | PathLike | None*) – If specified, replaces the specified "relative" directory, which by default is the parent directory.
- **tail** (*str | None*) – If specified, appends this text to the extension.
- **relative** (*str | PathLike | None*) – Replaces `relative` with `dpath` in `path`. Has no effect if `dpath` is not specified. Defaults to the dirname of the input `path`. *experimental* not currently implemented.
- **multidot** (*bool*) – Allows extensions to contain multiple dots. Specifically, if False, everything after the last dot in the basename is the extension. If True, everything after the first dot in the basename is the extension.

Returns augmented path

Return type *Path*

Example

```
>>> import ubelt as ub
>>> path = ub.Path('foo.bar')
>>> suffix = '_suff'
>>> prefix = 'pref_'
>>> ext = '.baz'
>>> newpath = path.augment(suffix, prefix, ext=ext, stem='bar')
>>> print('newpath = {!r}'.format(newpath))
newpath = Path('pref_bar_suff.baz')
```

delete()

Removes a file or recursively removes a directory. If a path does not exist, then this is does nothing.

SeeAlso: *ubelt.delete()*

Returns reference to self

Return type *Path*

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> base = ub.Path(ub.ensure_app_cache_dir('ubelt', 'delete_test2'))
>>> dpath1 = (base / 'dir').ensuredir()
>>> (base / 'dir' / 'subdir').ensuredir()
>>> (base / 'dir' / 'to_remove1.txt').touch()
>>> fpath1 = (base / 'dir' / 'subdir' / 'to_remove3.txt').touch()
>>> fpath2 = (base / 'dir' / 'subdir' / 'to_remove2.txt').touch()
>>> assert all(p.exists() for p in [dpath1, fpath1, fpath2])
>>> fpath1.delete()
>>> assert all(p.exists() for p in [dpath1, fpath2])
>>> assert not fpath1.exists()
>>> dpath1.delete()
>>> assert not any(p.exists() for p in [dpath1, fpath1, fpath2])
```

ensuredir(*mode=511*)

Concise alias of `self.mkdir(parents=True, exist_ok=True)`

Returns returns itself

Return type *Path*

Example

```
>>> import ubelt as ub
>>> cache_dpath = ub.ensure_app_cache_dir('ubelt')
>>> dpath = ub.Path(join(cache_dpath, 'ensuredir'))
>>> if dpath.exists():
...     os.rmdir(dpath)
>>> assert not dpath.exists()
>>> dpath.ensuredir()
>>> assert dpath.exists()
>>> dpath.rmdir()
```

expand()

Expands user tilde and environment variables.

Concise alias of *Path(os.path.expandvars(self.expanduser()))*

Returns path with expanded environment variables and tildes

Return type *Path*

Example

```
>>> import ubelt as ub
>>> #home_v1 = ub.Path('$HOME').expand()
>>> home_v2 = ub.Path('~/').expand()
>>> assert isinstance(home_v2, ub.Path)
>>> home_v3 = ub.Path.home()
>>> #print('home_v1 = {!r}'.format(home_v1))
>>> print('home_v2 = {!r}'.format(home_v2))
>>> print('home_v3 = {!r}'.format(home_v3))
>>> assert home_v3 == home_v2 # == home_v1
```

expandvars()

As discussed in [CPythonIssue21301], CPython won't be adding expandvars to pathlib. I think this is a mistake, so I added it in this extension.

Returns path with expanded environment variables

Return type *Path*

References

ls()

A convenience function to list all paths in a directory.

This is simply a wraper around iterdir that returns the results as a list instead of a generator. This is mainly for faster navigation in IPython. In production code *iterdir* should be used instead.

Returns List[Path]

Example

```
>>> import ubelt as ub
>>> self = ub.Path.appdir('ubelt/tests/ls')
>>> (self / 'dir1').ensuredir()
>>> (self / 'dir2').ensuredir()
>>> (self / 'file1').touch()
>>> (self / 'file2').touch()
>>> (self / 'dir1/file3').touch()
>>> (self / 'dir2/file4').touch()
>>> children = self.ls()
>>> assert isinstance(children, list)
>>> print(ub.repr2(sorted([p.relative_to(self) for p in children])))
[
    Path('dir1'),
    Path('dir2'),
    Path('file1'),
    Path('file2'),
]
```

shrinkuser(*home='~'*)

Inverse of `os.path.expanduser()`.

Parameters **home** (*str*) – symbol used to replace the home path. Defaults to '~', but you might want to use '$HOME' or '%USERPROFILE%' instead.

Returns path - shortened path replacing the home directory with a symbol

Return type *Path*

Example

```
>>> import ubelt as ub
>>> path = ub.Path('~').expand()
>>> assert str(path.shrinkuser()) == '~'
>>> assert str(ub.Path((str(path) + '1')).shrinkuser()) == str(path) + '1'
>>> assert str((path / '1').shrinkuser()) == join('~', '1')
>>> assert str((path / '1').shrinkuser('$HOME')) == join('$HOME', '1')
>>> assert str(ub.Path('.').shrinkuser()) == '.'
```

touch(*mode=438, exist_ok=True*)

Create this file with the given access mode, if it doesn't exist.

Returns returns itself

Return type *Path*

Notes

The `ubelt.util_io.touch()` function currently has a slightly different implementation. This uses whatever the pathlib version is. This may change in the future.

walk(*topdown=True*, *onerror=None*, *followlinks=False*)

A variant of os.walk for pathlib

Parameters

- **topdown** (*bool*) – if True starts yield nodes closer to the root first otherwise yield nodes closer to the leaves first.
- **onerror** (*Callable[[OSError], None]*) – A function with one argument of type OSError. If the error is raised the walk is aborted, otherwise it continues.
- **followlinks** (*bool*) – if True recurse into symbolic directory links

Yields *Tuple['Path', str, str]* – the root, file names, and directory names

Example

```
>>> import ubelt as ub
>>> self = ub.Path.appdir('ubelt/tests/ls')
>>> (self / 'dir1').ensuredir()
>>> (self / 'dir2').ensuredir()
>>> (self / 'file1').touch()
>>> (self / 'file2').touch()
>>> (self / 'dir1/file3').touch()
>>> (self / 'dir2/file4').touch()
>>> subdirs = list(self.walk())
>>> assert len(subdirs) == 3
```

class ubelt.**ProgIter**(*iterable=None*, *desc=None*, *total=None*, *freq=1*, *initial=0*, *eta_window=64*, *clearline=True*, *adjust=True*, *time_thresh=2.0*, *show_times=True*, *show_wall=False*, *enabled=True*, *verbose=None*, *stream=None*, *chunksize=None*, *rel_adjust_limit=4.0*, ***kwargs*)

Bases: `ubelt.progiter._TQDMCompat`, `ubelt.progiter._BackwardsCompat`

Prints progress as an iterator progresses

ProgIter is an alternative to *tqdm*. ProgIter implements much of the tqdm-API. The main difference between *ProgIter* and *tqdm* is that ProgIter does not use threading where as *tqdm* does.

Variables

- **iterable** (*iterable*) – An iterable iterable
- **desc** (*str*) – description label to show with progress
- **total** (*int*) – Maximum length of the process. If not specified, we estimate it from the iterable, if possible.
- **freq** (*int, default=1*) – How many iterations to wait between messages.
- **adjust** (*bool, default=True*) – if True freq is adjusted based on time_thresh
- **eta_window** (*int, default=64*) – number of previous measurements to use in eta calculation

- **clearline** (*bool, default=True*) – if True messages are printed on the same line otherwise each new progress message is printed on new line.
- **adjust** – if True *freq* is adjusted based on time_thresh. This may be overwritten depending on the setting of verbose.
- **time_thresh** (*float, default=2.0*) – desired amount of time to wait between messages if adjust is True otherwise does nothing
- **show_times** (*bool, default=True*) – shows rate and eta
- **show_wall** (*bool, default=False*) – show wall time
- **initial** (*int, default=0*) – starting index offset
- **stream** (*file, default=sys.stdout*) – stream where progress information is written to
- **enabled** (*bool, default=True*) – if False nothing happens.
- **chunksize** (*int, optional*) – indicates that each iteration processes a batch of this size. Iteration rate is displayed in terms of single-items.
- **rel_adjust_limit** (*float, default=4.0*) – Maximum factor update frequency can be adjusted by in a single step.
- **verbose** (*int*) – verbosity mode, which controls clearline, adjust, and enabled. The following maps the value of *verbose* to its effect. 0: enabled=False, 1: enabled=True with clearline=True and adjust=True, 2: enabled=True with clearline=False and adjust=True, 3: enabled=True with clearline=False and adjust=False

Note: Either use ProgIter in a with statement or call prog.end() at the end of the computation if there is a possibility that the entire iterable may not be exhausted.

Note: ProgIter is an alternative to *tqdm*. The main difference between *ProgIter* and *tqdm* is that ProgIter does not use threading where as *tqdm* does. *ProgIter* is simpler than *tqdm* and thus more stable in certain circumstances.

SeeAlso: tqdm - https://pypi.python.org/pypi/tqdm

References

http://datagenetics.com/blog/february12017/index.html

Example

```
>>>
>>> def is_prime(n):
...     return n >= 2 and not any(n % i == 0 for i in range(2, n))
>>> for n in ProgIter(range(100), verbose=1, show_wall=True):
>>>     # do some work
>>>     is_prime(n)
100/100... rate=... Hz, total=..., wall=...
```

set_extra(*extra*)

specify a custom info appended to the end of the next message

Todo:

- [ ] extra is a bad name; come up with something better and rename

Example

```
>>> prog = ProgIter(range(100, 300, 100), show_times=False, verbose=3)
>>> for n in prog:
>>>     prog.set_extra('processesing num {}'.format(n))
0/2...
1/2...processesing num 100
2/2...processesing num 200
```

step(*inc=1*, *force=False*)

Manually step progress update, either directly or by an increment.

Parameters

- **inc** (*int, default=1*) – number of steps to increment
- **force** (*bool, default=False*) – if True forces progress display

Example

```
>>> n = 3
>>> prog = ProgIter(desc='manual', total=n, verbose=3)
>>> # Need to manually begin and end in this mode
>>> prog.begin()
>>> for _ in range(n):
...     prog.step()
>>> prog.end()
```

Example

```
>>> n = 3
>>> # can be used as a context manager in manual mode
>>> with ProgIter(desc='manual', total=n, verbose=3) as prog:
...     for _ in range(n):
...         prog.step()
```

start()

Alias of *ubelt.progiter.ProgIter.begin()*

begin()

Initializes information used to measure progress

This only needs to be used if this ProgIter is not wrapping an iterable. Does nothing if the this ProgIter is disabled.

Returns a chainable self-reference

Return type *ProgIter*

end()

Signals that iteration has ended and displays the final message.

This only needs to be used if this ProgIter is not wrapping an iterable. Does nothing if the this ProgIter object is disabled or has already finished.

format_message()

builds a formatted progress message with the current values. This contains the special characters needed to clear lines.

Example

```
>>> self = ProgIter(clearline=False, show_times=False)
>>> print(repr(self.format_message()))
'    0/?... \n'
>>> self.begin()
>>> self.step()
>>> print(repr(self.format_message()))
' 1/?... \n'
```

Example

```
>>> self = ProgIter(chunksize=10, total=100, clearline=False,
>>>                 show_times=False, microseconds=True)
>>> # hack, microseconds=True for coverage, needs real test
>>> print(repr(self.format_message()))
' 0.00% of 10x100... \n'
>>> self.begin()
>>> self.update()  # tqdm alternative to step
>>> print(repr(self.format_message()))
' 1.00% of 10x100... \n'
```

ensure_newline()

use before any custom printing when using the progress iter to ensure your print statement starts on a new line instead of at the end of a progress line

Example

```
>>> # Unsafe version may write your message on the wrong line
>>> prog = ProgIter(range(3), show_times=False, freq=2, adjust=False)
>>> for n in prog:
...     print('unsafe message')
 0/3... unsafe message
unsafe message
 2/3... unsafe message
 3/3...
>>> # apparently the safe version does this too.
```

(continues on next page)

(continued from previous page)

```
>>> print('---')
---
>>> prog = ProgIter(range(3), show_times=False, freq=2, adjust=False)
>>> for n in prog:
...     prog.ensure_newline()
...     print('safe message')
 0/3...
safe message
safe message
 2/3...
safe message
 3/3...
```

display_message()

Writes current progress to the output stream

class ubelt.**TeeStringIO**(*redirect=None*)

Bases: `_io.StringIO`

An IO object that writes to itself and another IO stream.

Variables **redirect** (*io.IOBase*) – The other stream to write to.

Example

```
>>> import ubelt as ub
>>> redirect = io.StringIO()
>>> self = ub.TeeStringIO(redirect)
```

isatty()

Returns true of the redirect is a terminal.

Note: Needed for `IPython.embed` to work properly when this class is used to override stdout / stderr.

fileno()

Returns underlying file descriptor of the redirected IOBase object if one exists.

Example

```
>>> # Not sure the best way to test, this func is important for
>>> # capturing stdout when ipython embedding
>>> import pytest
>>> with pytest.raises(io.UnsupportedOperation):
>>>     TeeStringIO(redirect=io.StringIO()).fileno()
>>> with pytest.raises(io.UnsupportedOperation):
>>>     TeeStringIO(None).fileno()
```

property encoding

Gets the encoding of the *redirect* IO object

Example

```
>>> import ubelt as ub
>>> redirect = io.StringIO()
>>> assert ub.TeeStringIO(redirect).encoding is None
>>> assert ub.TeeStringIO(None).encoding is None
>>> assert ub.TeeStringIO(sys.stdout).encoding is sys.stdout.encoding
>>> redirect = io.TextIOWrapper(io.StringIO())
>>> assert ub.TeeStringIO(redirect).encoding is redirect.encoding
```

write(*msg*)

Write to this and the redirected stream

flush()

Flush to this and the redirected stream

class ubelt.**TempDir**

Bases: `object`

Context for creating and cleaning up temporary directories.

Note: This class will be DEPRECATED. The exact deprecation version and mitigation plan has not yet been developed.

Note: This exists because `tempfile.TemporaryDirectory` was introduced in Python 3.2. Thus once ubelt no longer supports python 2.7, this class will be deprecated.

Example

```
>>> from ubelt.util_path import *  # NOQA
>>> with TempDir() as self:
>>>     dpath = self.dpath
>>>     assert exists(dpath)
>>> assert not exists(dpath)
```

Example

```
>>> from ubelt.util_path import *  # NOQA
>>> self = TempDir()
>>> dpath = self.ensure()
>>> assert exists(dpath)
>>> self.cleanup()
>>> assert not exists(dpath)
```

ensure()

cleanup()

start()

class ubelt.**Timer**(*label=''*, *verbose=None*, *newline=True*)

Bases: object

Measures time elapsed between a start and end point. Can be used as a with-statement context manager, or using the tic/toc api.

Parameters

- **label** (*str, default=''*) – identifier for printing
- **verbose** (*int, default=None*) – verbosity flag, defaults to True if label is given
- **newline** (*bool, default=True*) – if False and verbose, print tic and toc on the same line

Variables

- **elapsed** (*float*) – number of seconds measured by the context manager
- **tstart** (*float*) – time of last *tic* reported by *self._time()*

Example

```
>>> # Create and start the timer using the context manager
>>> import math
>>> timer = Timer('Timer test!', verbose=1)
>>> with timer:
>>>     math.factorial(10)
>>> assert timer.elapsed > 0
tic('Timer test!')
...toc('Timer test!')=...
```

Example

```
>>> # Create and start the timer using the tic/toc interface
>>> timer = Timer().tic()
>>> elapsed1 = timer.toc()
>>> elapsed2 = timer.toc()
>>> elapsed3 = timer.toc()
>>> assert elapsed1 <= elapsed2
>>> assert elapsed2 <= elapsed3
```

tic()

starts the timer

toc()

stops the timer

ubelt.**allsame**(*iterable*, *eq=<built-in function eq>*)

Determine if all items in a sequence are the same

Parameters

- **iterable** (*Iterable[T]*) – items to determine if they are all the same
- **eq** (*Callable[[T, T], bool], default=operator.eq*) – function used to test for equality

Returns True if all items are equal, otherwise False

Return type bool

Notes

Similar to `more_itertools.all_equal()`

Example

```
>>> import ubelt as ub
>>> ub.allsame([1, 1, 1, 1])
True
>>> ub.allsame([])
True
>>> ub.allsame([0, 1])
False
>>> iterable = iter([0, 1, 1, 1])
>>> next(iterable)
>>> ub.allsame(iterable)
True
>>> ub.allsame(range(10))
False
>>> ub.allsame(range(10), lambda a, b: True)
True
```

`ubelt.argflag`(*key*, *argv=None*)

Determines if a key is specified on the command line.

This is a functional alternative to `key in sys.argv`.

Parameters

- **key** (*str | Tuple[str, ...]*) – string or tuple of strings. Each key should be prefixed with two hyphens (i.e. `--`).
- **argv** (*List[str], default=None*) – overrides `sys.argv` if specified

Returns flag - True if the key (or any of the keys) was specified

Return type bool

Example

```
>>> import ubelt as ub
>>> argv = ['--spam', '--eggs', 'foo']
>>> assert ub.argflag('--eggs', argv=argv) is True
>>> assert ub.argflag('--ans', argv=argv) is False
>>> assert ub.argflag('foo', argv=argv) is True
>>> assert ub.argflag(('bar', '--spam'), argv=argv) is True
```

`ubelt.argmax`(*indexable*, *key=None*)

Returns index / key of the item with the largest value.

This is similar to `numpy.argmax()`, but it is written in pure python and works on both lists and dictionaries.

Parameters

- **indexable** (*Iterable[VT] | Mapping[KT, VT]*) – indexable to sort by
- **key** (*Callable[[VT], Any], default=None*) – customizes the ordering of the indexable

Returns the index of the item with the maximum value.

Return type int | KT

Example

```
>>> import ubelt as ub
>>> assert ub.argmax({'a': 3, 'b': 2, 'c': 100}) == 'c'
>>> assert ub.argmax(['a', 'c', 'b', 'z', 'f']) == 3
>>> assert ub.argmax([[0, 1], [2, 3, 4], [5]], key=len) == 1
>>> assert ub.argmax({'a': 3, 'b': 2, 3: 100, 4: 4}) == 3
>>> assert ub.argmax(iter(['a', 'c', 'b', 'z', 'f'])) == 3
```

`ubelt.argmin(indexable, key=None)`

Returns index / key of the item with the smallest value.

This is similar to `numpy.argmin()`, but it is written in pure python and works on both lists and dictionaries.

Parameters

- **indexable** (*Iterable[VT] | Mapping[KT, VT]*) – indexable to sort by
- **key** (*Callable[[VT], VT], default=None*) – customizes the ordering of the indexable

Returns the index of the item with the minimum value.

Return type int | KT

Example

```
>>> import ubelt as ub
>>> assert ub.argmin({'a': 3, 'b': 2, 'c': 100}) == 'b'
>>> assert ub.argmin(['a', 'c', 'b', 'z', 'f']) == 0
>>> assert ub.argmin([[0, 1], [2, 3, 4], [5]], key=len) == 2
>>> assert ub.argmin({'a': 3, 'b': 2, 3: 100, 4: 4}) == 'b'
>>> assert ub.argmin(iter(['a', 'c', 'A', 'z', 'f'])) == 2
```

`ubelt.argsort(indexable, key=None, reverse=False)`

Returns the indices that would sort a indexable object.

This is similar to `numpy.argsort()`, but it is written in pure python and works on both lists and dictionaries.

Parameters

- **indexable** (*Iterable[VT] | Mapping[KT, VT]*) – indexable to sort by
- **key** (*Callable[[VT], VT] | None, default=None*) – customizes the ordering of the indexable
- **reverse** (*bool, default=False*) – if True returns in descending order

Returns indices - list of indices that sorts the indexable

Return type List[int] | List[KT]

Example

```
>>> import ubelt as ub
>>> # argsort works on dicts by returning keys
>>> dict_ = {'a': 3, 'b': 2, 'c': 100}
>>> indices = ub.argsort(dict_)
>>> assert list(ub.take(dict_, indices)) == sorted(dict_.values())
>>> # argsort works on lists by returning indices
>>> indexable = [100, 2, 432, 10]
>>> indices = ub.argsort(indexable)
>>> assert list(ub.take(indexable, indices)) == sorted(indexable)
>>> # Can use iterators, but be careful. It exhausts them.
>>> indexable = reversed(range(100))
>>> indices = ub.argsort(indexable)
>>> assert indices[0] == 99
>>> # Can use key just like sorted
>>> indexable = [[0, 1, 2], [3, 4], [5]]
>>> indices = ub.argsort(indexable, key=len)
>>> assert indices == [2, 1, 0]
>>> # Can use reverse just like sorted
>>> indexable = [0, 2, 1]
>>> indices = ub.argsort(indexable, reverse=True)
>>> assert indices == [1, 2, 0]
```

`ubelt.argunique`(*items*, *key=None*)

Returns indices corresponding to the first instance of each unique item.

Parameters

- **items** (*Sequence[VT]*) – indexable collection of items
- **key** (*Callable[[VT], Any], default=None*) – custom normalization function. If specified returns items where `key(item)` is unique.

Returns indices of the unique items

Return type Iterator[int]

Example

```
>>> import ubelt as ub
>>> items = [0, 2, 5, 1, 1, 0, 2, 4]
>>> indices = list(ub.argunique(items))
>>> assert indices == [0, 1, 2, 3, 7]
>>> indices = list(ub.argunique(items, key=lambda x: x % 2 == 0))
>>> assert indices == [0, 2]
```

`ubelt.argval`(*key*, *default=NoParam*, *argv=None*)

Get the value of a keyword argument specified on the command line.

Values can be specified as `<key> <value>` or `<key>=<value>`

Parameters

- **key** (*str | Tuple[str, ...]*) – string or tuple of strings. Each key should be prefixed with two hyphens (i.e. `--`)

- **default** (*T | util_const._NoParamType, default=NoParam*) – a value to return if not specified.
- **argv** (*Optional[List[str]], default=None*) – uses `sys.argv` if unspecified

Returns value - the value specified after the key. It they key is specified multiple times, then the first value is returned.

Return type str | T

Todo:

- [ ] Can we handle the case where the value is a list of long paths?
- [ ] Should we default the first or last specified instance of the flag.

Example

```
>>> import ubelt as ub
>>> argv = ['--ans', '42', '--quest=the grail', '--ans=6', '--bad']
>>> assert ub.argval('--spam', argv=argv) == ub.NoParam
>>> assert ub.argval('--quest', argv=argv) == 'the grail'
>>> assert ub.argval('--ans', argv=argv) == '42'
>>> assert ub.argval('--bad', argv=argv) == ub.NoParam
>>> assert ub.argval(('--bad', '--bar'), argv=argv) == ub.NoParam
```

Example

```
>>> # Test fix for GH Issue #41
>>> import ubelt as ub
>>> argv = ['--path=/path/with/k=3']
>>> ub.argval('--path', argv=argv) == '/path/with/k=3'
```

`ubelt.augpath`(*path*, *suffix=''*, *prefix=''*, *ext=None*, *tail=''*, *base=None*, *dpath=None*, *relative=None*, *multidot=False*)

Create a new path with a different extension, basename, directory, prefix, and/or suffix.

A prefix is inserted before the basename. A suffix is inserted between the basename and the extension. The basename and extension can be replaced with a new one. Essentially a path is broken down into components (dpath, base, ext), and then recombined as (dpath, prefix, base, suffix, ext) after replacing any specified component.

Parameters

- **path** (*str | PathLike*) – a path to augment
- **suffix** (*str*) – placed between the basename and extension
- **prefix** (*str*) – placed in front of the basename
- **ext** (*str | None*) – if specified, replaces the extension
- **tail** (*str | None*) – If specified, appends this text to the extension
- **base** (*str | None*) – if specified, replaces the basename without extension. Note: this is referred to as stem in `ub.Path`.
- **dpath** (*str | PathLike | None*) – if specified, replaces the specified "relative" directory, which by default is the parent directory.

- **relative** (*str | PathLike | None*) – Replaces `relative` with `dpath` in `path`. Has no effect if `dpath` is not specified. Defaults to the dirname of the input `path`. *experimental* not currently implemented.
- **multidot** (*bool*) – Allows extensions to contain multiple dots. Specifically, if False, everything after the last dot in the basename is the extension. If True, everything after the first dot in the basename is the extension.

Returns augmented path

Return type str

Example

```
>>> import ubelt as ub
>>> path = 'foo.bar'
>>> suffix = '_suff'
>>> prefix = 'pref_'
>>> ext = '.baz'
>>> newpath = ub.augpath(path, suffix, prefix, ext=ext, base='bar')
>>> print('newpath = %s' % (newpath,))
newpath = pref_bar_suff.baz
```

Example

```
>>> from ubelt.util_path import *  # NOQA
>>> augpath('foo.bar')
'foo.bar'
>>> augpath('foo.bar', ext='.BAZ')
'foo.BAZ'
>>> augpath('foo.bar', suffix='_')
'foo_.bar'
>>> augpath('foo.bar', prefix='_')
'_foo.bar'
>>> augpath('foo.bar', base='baz')
'baz.bar'
>>> augpath('foo.tar.gz', ext='.zip', multidot=True)
foo.zip
>>> augpath('foo.tar.gz', ext='.zip', multidot=False)
foo.tar.zip
>>> augpath('foo.tar.gz', suffix='_new', multidot=True)
foo_new.tar.gz
>>> augpath('foo.tar.gz', suffix='_new', tail='.cache', multidot=True)
foo_new.tar.gz.cache
```

`ubelt.`**`boolmask`**(*indices*, *maxval=None*)

Constructs a list of booleans where an item is True if its position is in `indices` otherwise it is False.

Parameters

- **indices** (*List[int]*) – list of integer indices
- **maxval** (*int*) – length of the returned list. If not specified this is inferred using `max(indices)`

Returns mask - a list of booleans. mask[idx] is True if idx in indices

Return type List[bool]

Note: In the future the arg `maxval` may change its name to `shape`

Example

```
>>> import ubelt as ub
>>> indices = [0, 1, 4]
>>> mask = ub.boolmask(indices, maxval=6)
>>> assert mask == [True, True, False, False, True, False]
>>> mask = ub.boolmask(indices)
>>> assert mask == [True, True, False, False, True]
```

class `ubelt.`**`chunks`**(*items*, *chunksize=None*, *nchunks=None*, *total=None*, *bordermode='none'*, *legacy=False*)

Bases: `object`

Generates successive n-sized chunks from `items`.

If the last chunk has less than n elements, `bordermode` is used to determine fill values.

Parameters

- **items** (*Iterable[T]*) – input to iterate over
- **chunksize** (*int*) – size of each sublist yielded
- **nchunks** (*int*) – number of chunks to create (cannot be specified if chunksize is specified)
- **bordermode** (*str*) – determines how to handle the last case if the length of the input is not divisible by chunksize valid values are: {'none', 'cycle', 'replicate'}
- **total** (*int*) – hints about the length of the input

Note:

FIXME: When nchunks is given, that's how many chunks we should get but the issue is that chunksize is not well defined in that instance For instance how do we turn a list with 4 elements into 3 chunks where does the extra item go?

In ubelt <= 0.10.3 there is a bug when specifying nchunks, where it chooses a chunksize that is too large. Specify `legacy=True` to get the old buggy behavior if needed.

Notes

This is similar to functionality provided by `more_itertools.chunked()`, `more_itertools.chunked_even()`, `more_itertools.sliced()`, `more_itertools.divide()`,

Yields *List[T]* – subsequent non-overlapping chunks of the input items

References

Example

```
>>> import ubelt as ub
>>> items = '1234567'
>>> genresult = ub.chunks(items, chunksize=3)
>>> list(genresult)
[['1', '2', '3'], ['4', '5', '6'], ['7']]
```

Example

```
>>> import ubelt as ub
>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> genresult = ub.chunks(items, chunksize=3, bordermode='none')
>>> assert list(genresult) == [[1, 2, 3], [4, 5, 6], [7]]
>>> genresult = ub.chunks(items, chunksize=3, bordermode='cycle')
>>> assert list(genresult) == [[1, 2, 3], [4, 5, 6], [7, 1, 2]]
>>> genresult = ub.chunks(items, chunksize=3, bordermode='replicate')
>>> assert list(genresult) == [[1, 2, 3], [4, 5, 6], [7, 7, 7]]
```

Example

```
>>> import ubelt as ub
>>> assert len(list(ub.chunks(range(2), nchunks=2))) == 2
>>> assert len(list(ub.chunks(range(3), nchunks=2))) == 2
>>> # Note: ub.chunks will not do the 2,1,1 split
>>> assert len(list(ub.chunks(range(4), nchunks=3))) == 3
>>> assert len(list(ub.chunks([], 2, bordermode='none'))) == 0
>>> assert len(list(ub.chunks([], 2, bordermode='cycle'))) == 0
>>> assert len(list(ub.chunks([], 2, None, bordermode='replicate'))) == 0
```

Example

```
>>> from ubelt.util_list import *  # NOQA
>>> def _check_len(self):
...     assert len(self) == len(list(self))
>>> _check_len(chunks(list(range(3)), nchunks=2))
>>> _check_len(chunks(list(range(2)), nchunks=2))
>>> _check_len(chunks(list(range(2)), nchunks=3))
```

Example

```
>>> from ubelt.util_list import *  # NOQA
>>> import pytest
>>> assert pytest.raises(ValueError, chunks, range(9))
>>> assert pytest.raises(ValueError, chunks, range(9), chunksize=2, nchunks=2)
>>> assert pytest.raises(TypeError, len, chunks((_ for _ in range(2)), 2))
```

Example

```
>>> from ubelt.util_list import *  # NOQA
>>> import ubelt as ub
>>> basis = {
>>>     'legacy': [False, True],
>>>     'chunker': [{'nchunks': 3}, {'nchunks': 4}, {'nchunks': 5}, {'nchunks': 7},
↪{'chunksize': 3}],
>>>     'items': [range(2), range(4), range(5), range(7), range(9)],
>>>     'bordermode': ['none', 'cycle', 'replicate'],
>>> }
>>> grid_items = list(ub.named_product(basis))
>>> rows = []
>>> for grid_item in ub.ProgIter(grid_items):
>>>     chunker = grid_item.get('chunker')
>>>     grid_item.update(chunker)
>>>     kw = ub.dict_diff(grid_item, {'chunker'})
>>>     self = chunk_iter = ub.chunks(**kw)
>>>     chunked = list(chunk_iter)
>>>     chunk_lens = list(map(len, chunked))
>>>     row = ub.dict_union(grid_item, {'chunk_lens': chunk_lens, 'chunks': chunked}
↪)
>>>     row['chunker'] = str(row['chunker'])
>>>     if not row['legacy'] and 'nchunks' in kw:
>>>         assert kw['nchunks'] == row['nchunks']
>>>     row.update(chunk_iter.__dict__)
>>>     rows.append(row)
>>> # xdoctest: +SKIP
>>> import pandas as pd
>>> df = pd.DataFrame(rows)
>>> for _, subdf in df.groupby('chunker'):
>>>     print(subdf)
```

static noborder(*items*, *chunksize*)

static cycle(*items*, *chunksize*)

static replicate(*items*, *chunksize*)

ubelt.**cmd**(*command*, *shell=False*, *detach=False*, *verbose=0*, *tee=None*, *cwd=None*, *env=None*, *tee_backend='auto'*, *check=False*, *system=False*, *timeout=None*)

Executes a command in a subprocess.

The advantage of this wrapper around subprocess is that (1) you control if the subprocess prints to stdout, (2) the text written to stdout and stderr is returned for parsing, (3) cross platform behavior that lets you specify the

command as a string or tuple regardless of whether or not shell=True. (4) ability to detach, return the process object and allow the process to run in the background (eventually we may return a Future object instead).

Parameters

- **command** (*str | List[str]*) – bash-like command string or tuple of executable and args
- **shell** (*bool, default=False*) – if True, process is run in shell.
- **detach** (*bool, default=False*) – if True, process is detached and run in background.
- **verbose** (*int, default=0*) – verbosity mode. Can be 0, 1, 2, or 3.
- **tee** (*bool | None*) – if True, simultaneously writes to stdout while capturing output from the command. If not specified, defaults to True if verbose > 0. If detach is True, then this argument is ignored.
- **cwd** (*str | PathLike | None*) – Path to run command. Defaults to current working directory if unspecified.
- **env** (*Dict[str, str] | None*) – environment passed to Popen
- **tee_backend** (*str, default='auto'*) – backend for tee output. Valid choices are: "auto", "select" (POSIX only), and "thread".
- **check** (*bool, default=False*) – if True, check that the return code was zero before returning, otherwise raise a CalledProcessError. Does nothing if detach is True.
- **system** (*bool, default=False*) – if True, most other considerations are dropped, and `os.system()` is used to execute the command in a platform dependant way. Other arguments such as env, tee, timeout, and shell are all ignored. (new in version 1.1.0)
- **timeout** (*float*) – If the process does not complete in *timeout* seconds, raises a `subprocess.TimeoutExpired`. (new in version 1.1.0)

Returns info - information about command status. if detach is False `info` contains captured standard out, standard error, and the return code if detach is False `info` contains a reference to the process.

Return type dict

Note: Inputs can either be text or tuple based. On UNIX we ensure conversion to text if shell=True, and to tuple if shell=False. On windows, the input is always text based. See [SO_33560364] for a potential cross-platform shlex solution for windows.

When using the tee output, the stdout and stderr may be shuffled from what they would be on the command line.

CommandLine

```
xdoctest -m ubelt.util_cmd cmd:6
python -c "import ubelt as ub; ub.cmd('ping localhost -c 2', verbose=2)"
pytest "$(python -c 'import ubelt; print(ubelt.util_cmd.__file__)')" -sv --xdoctest-
↪verbose 2
```

References

Example

```
>>> import ubelt as ub
>>> info = ub.cmd(('echo', 'simple cmdline interface'), verbose=1)
simple cmdline interface
>>> assert info['ret'] == 0
>>> assert info['out'].strip() == 'simple cmdline interface'
>>> assert info['err'].strip() == ''
```

Example

```
>>> import ubelt as ub
>>> info = ub.cmd('echo str noshell', verbose=0)
>>> assert info['out'].strip() == 'str noshell'
```

Example

```
>>> # windows echo will output extra single quotes
>>> import ubelt as ub
>>> info = ub.cmd(('echo', 'tuple noshell'), verbose=0)
>>> assert info['out'].strip().strip("'") == 'tuple noshell'
```

Example

```
>>> # Note this command is formatted to work on win32 and unix
>>> import ubelt as ub
>>> info = ub.cmd('echo str&&echo shell', verbose=0, shell=True)
>>> assert info['out'].strip() == 'str' + chr(10) + 'shell'
```

Example

```
>>> import ubelt as ub
>>> info = ub.cmd(('echo', 'tuple shell'), verbose=0, shell=True)
>>> assert info['out'].strip().strip("'") == 'tuple shell'
```

Example

```
>>> import pytest
>>> import ubelt as ub
>>> info = ub.cmd('echo hi', check=True)
>>> import subprocess
>>> with pytest.raises(subprocess.CalledProcessError):
>>>     ub.cmd('exit 1', check=True, shell=True)
```

Example

```
>>> import ubelt as ub
>>> from os.path import join, exists
>>> fpath1 = join(ub.get_app_cache_dir('ubelt'), 'cmdout1.txt')
>>> fpath2 = join(ub.get_app_cache_dir('ubelt'), 'cmdout2.txt')
>>> ub.delete(fpath1)
>>> ub.delete(fpath2)
>>> # Start up two processes that run simultaneously in the background
>>> info1 = ub.cmd(('touch', fpath1), detach=True)
>>> info2 = ub.cmd('echo writing2 > ' + fpath2, shell=True, detach=True)
>>> # Detached processes are running in the background
>>> # We can run other code while we wait for them.
>>> while not exists(fpath1):
...     pass
>>> while not exists(fpath2):
...     pass
>>> # communicate with the process before you finish
>>> # (otherwise you may leak a text wrapper)
>>> info1['proc'].communicate()
>>> info2['proc'].communicate()
>>> # Check that the process actually did finish
>>> assert (info1['proc'].wait()) == 0
>>> assert (info2['proc'].wait()) == 0
>>> # Check that the process did what we expect
>>> assert ub.readfrom(fpath1) == ''
>>> assert ub.readfrom(fpath2).strip() == 'writing2'
```

Example

```
>>> # Can also use ub.cmd to call os.system
>>> import pytest
>>> import ubelt as ub
>>> import subprocess
>>> info = ub.cmd('echo hi', check=True, system=True)
>>> with pytest.raises(subprocess.CalledProcessError):
>>>     ub.cmd('exit 1', check=True, shell=True)
```

ubelt.**codeblock**(*text*)

Create a block of text that preserves all newlines and relative indentation

Wraps multiline string blocks and returns unindented code. Useful for templated code defined in indented parts of code.

Parameters **text** (*str*) – typically a multiline string

Returns the unindented string

Return type str

Example

```
>>> import ubelt as ub
>>> # Simulate an indented part of code
>>> if True:
>>>     # notice the indentation on this will be normal
>>>     codeblock_version = ub.codeblock(
...             '''
...             def foo():
...                 return 'bar'
...             '''
...         )
>>>     # notice the indentation and newlines on this will be odd
>>>     normal_version = ('''
...         def foo():
...             return 'bar'
...     ''')
>>> assert normal_version != codeblock_version
>>> print('Without codeblock')
>>> print(normal_version)
>>> print('With codeblock')
>>> print(codeblock_version)
```

ubelt.**color_text**(*text*, *color*)

Colorizes text a single color using ansi tags.

Parameters

- **text** (*str*) – text to colorize
- **color** (*str*) – color code. different systems may have different colors. commonly available colors are: 'red', 'brown', 'yellow', 'green', 'blue', 'black', and 'white'.

Returns text - colorized text. If pygments is not installed plain text is returned.

Return type str

Example

```
>>> text = 'raw text'
>>> import pytest
>>> import ubelt as ub
>>> if ub.modname_to_modpath('pygments'):
>>>     # Colors text only if pygments is installed
>>>     ansi_text = ub.ensure_unicode(color_text(text, 'red'))
>>>     prefix = ub.ensure_unicode('\x1b[31')
>>>     print('prefix = {!r}'.format(prefix))
>>>     print('ansi_text = {!r}'.format(ansi_text))
>>>     assert ansi_text.startswith(prefix)
>>>     assert color_text(text, None) == 'raw text'
>>> else:
>>>     # Otherwise text passes through unchanged
>>>     assert color_text(text, 'red') == 'raw text'
>>>     assert color_text(text, None) == 'raw text'
```

`ubelt.compatible(config, func, start=0)`

Take the subset of dict items that can be passed to function as kwargs

Parameters

- **config** (*Dict[str, Any]*) – a flat configuration dictionary
- **func** (*Callable*) – a function or method
- **start** (*int, default=0*) – Only take args after this position. Set to 1 if calling with an unbound method to avoid the `self` argument.

Returns a subset of `config` that only contains items compatible with the signature of `func`.

Return type Dict[str, Any]

Example

```
>>> # An example use case is to select a subset of of a config
>>> # that can be passed to some function as kwargs
>>> import ubelt as ub
>>> # Define a function with args that match some keys in a config.
>>> def func(a, e, f):
>>>     return a * e * f
>>> # Define a config that has a superset of items needed by the func
>>> config = {
...   'a': 2, 'b': 3, 'c': 7,
...   'd': 11, 'e': 13, 'f': 17,
... }
>>> # Call the function only with keys that are compatible
>>> func(**ub.compatible(config, func))
442
```

Example

```
>>> # Test case with kwargs
>>> import ubelt as ub
>>> def func(a, e, f, *args, **kwargs):
>>>     return a * e * f
>>> config = {
...   'a': 2, 'b': 3, 'c': 7,
...   'd': 11, 'e': 13, 'f': 17,
... }
>>> func(**ub.compatible(config, func))
```

`ubelt.compress(items, flags)`

Selects from `items` where the corresponding value in `flags` is True.

Parameters

- **items** (*Iterable[Any]*) – a sequence to select items from
- **flags** (*Iterable[bool]*) – corresponding sequence of bools

Returns a subset of masked items

Return type Iterable[Any]

Notes

This function is based on `numpy.compress()`, but is pure Python and swaps the condition and array argument to be consistent with *`ubelt.take()`*.

This is equivalent to `itertools.compress()`.

Example

```
>>> import ubelt as ub
>>> items = [1, 2, 3, 4, 5]
>>> flags = [False, True, True, False, True]
>>> list(ub.compress(items, flags))
[2, 3, 5]
```

`ubelt.`**`ddict`**

alias of `collections.defaultdict`

`ubelt.`**`delete`**(*path*, *verbose=False*)

Removes a file or recursively removes a directory. If a path does not exist, then this is does nothing.

Parameters

- **path** (*str | PathLike*) – file or directory to remove
- **verbose** (*bool*) – if True prints what is being done

SeeAlso:

send2trash - A cross-platform Python package for sending files to the trash instead of irreversibly deleting them.

`ubelt.util_path.Path.delete()`

Notes

This can call `os.unlink()`, `os.rmdir()`, or `shutil.rmtree()`, depending on what `path` references on the filesystem. (On windows may also call a custom `ubelt._win32_links._win32_rmtree()`).

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> base = ub.ensure_app_cache_dir('ubelt', 'delete_test')
>>> dpath1 = ub.ensuredir(join(base, 'dir'))
>>> ub.ensuredir(join(base, 'dir', 'subdir'))
>>> ub.touch(join(base, 'dir', 'to_remove1.txt'))
>>> fpath1 = join(base, 'dir', 'subdir', 'to_remove3.txt')
>>> fpath2 = join(base, 'dir', 'subdir', 'to_remove2.txt')
>>> ub.touch(fpath1)
>>> ub.touch(fpath2)
>>> assert all(map(exists, (dpath1, fpath1, fpath2)))
>>> ub.delete(fpath1)
```

(continues on next page)

(continued from previous page)

```
>>> assert all(map(exists, (dpath1, fpath2)))
>>> assert not exists(fpath1)
>>> ub.delete(dpath1)
>>> assert not any(map(exists, (dpath1, fpath1, fpath2)))
```

Example

```
>>> import ubelt as ub
>>> from os.path import exists, join
>>> dpath = ub.ensure_app_cache_dir('ubelt', 'delete_test2')
>>> dpath1 = ub.ensuredir(join(dpath, 'dir'))
>>> fpath1 = ub.touch(join(dpath1, 'to_remove.txt'))
>>> assert exists(fpath1)
>>> ub.delete(dpath)
>>> assert not exists(fpath1)
```

`ubelt.dict_diff(*args)`

Dictionary set extension for `set.difference()`

Constructs a dictionary that contains any of the keys in the first arg, which are not in any of the following args.

Parameters ***args** (*List[Dict[KT, VT] | Iterable[KT]]*) – A sequence of dictionaries (or sets of keys). The first argument should always be a dictionary, but the subsequent arguments can just be sets of keys.

Returns OrderedDict if the first argument is an OrderedDict, otherwise dict

Return type Dict[KT, VT] | OrderedDict[KT, VT]

Todo:

- [ ] Add inplace keyword argument, which modifies the first dictionary inplace.

Example

```
>>> import ubelt as ub
>>> ub.dict_diff({'a': 1, 'b': 1}, {'a'}, {'c'})
{'b': 1}
>>> ub.dict_diff(odict([('a', 1), ('b', 2)]), odict([('c', 3)]))
OrderedDict([('a', 1), ('b', 2)])
>>> ub.dict_diff()
{}
>>> ub.dict_diff({'a': 1, 'b': 2}, {'c'})
```

`ubelt.dict_hist(items, weights=None, ordered=False, labels=None)`

Builds a histogram of items, counting the number of time each item appears in the input.

Parameters

- **items** (*Iterable[T]*) – hashable items (usually containing duplicates)
- **weights** (*Iterable[float], default=None*) – Corresponding weights for each item.

- **ordered** (*bool, default=False*) – If True the result is ordered by frequency.
- **labels** (*Iterable[T], default=None*) – Expected labels. Allows this function to pre-initialize the histogram. If specified the frequency of each label is initialized to zero and `items` can only contain items specified in labels.

Returns dictionary where the keys are unique elements from `items`, and the values are the number of times the item appears in `items`.

Return type dict[T, int]

Example

```
>>> import ubelt as ub
>>> items = [1, 2, 39, 900, 1232, 900, 1232, 2, 2, 2, 900]
>>> hist = ub.dict_hist(items)
>>> print(ub.repr2(hist, nl=0))
{1: 1, 2: 4, 39: 1, 900: 3, 1232: 2}
```

Example

```
>>> import ubelt as ub
>>> items = [1, 2, 39, 900, 1232, 900, 1232, 2, 2, 2, 900]
>>> hist1 = ub.dict_hist(items)
>>> hist2 = ub.dict_hist(items, ordered=True)
>>> try:
>>>     hist3 = ub.dict_hist(items, labels=[])
>>> except KeyError:
>>>     pass
>>> else:
>>>     raise AssertionError('expected key error')
>>> weights = [1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1]
>>> hist4 = ub.dict_hist(items, weights=weights)
>>> print(ub.repr2(hist1, nl=0))
{1: 1, 2: 4, 39: 1, 900: 3, 1232: 2}
>>> print(ub.repr2(hist4, nl=0))
{1: 1, 2: 4, 39: 1, 900: 1, 1232: 0}
```

`ubelt.`**`dict_isect`**(**args*)

Dictionary set extension for `set.intersection()`

Constructs a dictionary that contains keys common between all inputs. The returned values will only belong to the first dictionary.

Parameters ***args** (*List[Dict[KT, VT] | Iterable[KT]]*) – A sequence of dictionaries (or sets of keys). The first argument should always be a dictionary, but the subsequent arguments can just be sets of keys.

Returns OrderedDict if the first argument is an OrderedDict, otherwise dict

Return type Dict[KT, VT] | OrderedDict[KT, VT]

Note: This function can be used as an alternative to *`dict_subset()`* where any key not in the dictionary is ignored. See the following example:

```
>>> import ubelt as ub
>>> # xdoctest: +IGNORE_WANT
>>> ub.dict_isect({'a': 1, 'b': 2, 'c': 3}, ['a', 'c', 'd'])
{'a': 1, 'c': 3}
```

Example

```
>>> import ubelt as ub
>>> ub.dict_isect({'a': 1, 'b': 1}, {'b': 2, 'c': 2})
{'b': 1}
>>> ub.dict_isect(odict([('a', 1), ('b', 2)]), odict([('c', 3)]))
OrderedDict()
>>> ub.dict_isect()
{}
```

ubelt.**dict_subset**(*dict_*, *keys*, *default=NoParam*, *cls=<class 'collections.OrderedDict'>*)

Get a subset of a dictionary

Parameters

- **dict_** (*Dict[KT, VT]*) – superset dictionary
- **keys** (*Iterable[KT]*) – keys to take from `dict_`
- **default** (*Optional[object] | util_const._NoParamType*) – if specified uses default if keys are missing.
- **cls** (*Type[Dict], default=OrderedDict*) – type of the returned dictionary.

Returns subset dictionary

Return type Dict[KT, VT]

SeeAlso: *dict_isect()* - similar functionality, but ignores missing keys

Example

```
>>> import ubelt as ub
>>> dict_ = {'K': 3, 'dcvs_clip_max': 0.2, 'p': 0.1}
>>> keys = ['K', 'dcvs_clip_max']
>>> subdict_ = ub.dict_subset(dict_, keys)
>>> print(ub.repr2(subdict_, nl=0))
{'K': 3, 'dcvs_clip_max': 0.2}
```

ubelt.**dict_union**(**args*)

Dictionary set extension for `set.union`

Combines items with from multiple dictionaries. For items with intersecting keys, dictionaries towards the end of the sequence are given precedence.

Parameters ***args** (*List[Dict]*) – A sequence of dictionaries. Values are taken from the last

Returns OrderedDict if the first argument is an OrderedDict, otherwise dict

Return type Dict | OrderedDict

Notes

In Python 3.8+, the bitwise or operator "|" operator performs a similar operation, but as of 2022-06-01 there is still no public method for dictionary union (or any other dictionary set operator).

References

https://stackoverflow.com/questions/38987/merge-two-dict

SeeAlso: `collections.ChainMap()` - a standard python builtin data structure that provides a view that treats multiple dicts as a single dict. https://docs.python.org/3/library/collections.html#chainmap-objects

Example

```
>>> import ubelt as ub
>>> result = ub.dict_union({'a': 1, 'b': 1}, {'b': 2, 'c': 2})
>>> assert result == {'a': 1, 'b': 2, 'c': 2}
>>> ub.dict_union(
>>>     ub.odict([('a', 1), ('b', 2)]),
>>>     ub.odict([('c', 3), ('d', 4)]))
OrderedDict([('a', 1), ('b', 2), ('c', 3), ('d', 4)])
>>> ub.dict_union()
{}
```

`ubelt.download`(*url*, *fpath=None*, *dpath=None*, *fname=None*, *hash_prefix=None*, *hasher='sha512'*, *chunksize=8192*, *verbose=1*, *timeout=NoParam*, *progkw=None*)

Downloads a url to a file on disk.

If unspecified the location and name of the file is chosen automatically. A hash_prefix can be specified to verify the integrity of the downloaded data. This function will download the data every time its called. For cached downloading see *grabdata*.

Parameters

- **url** (*str*) – The url to download.
- **fpath** (*Optional[str | PathLike | io.BytesIO]*) – The path to download to. Defaults to basename of url and ubelt's application cache. If this is a io.BytesIO object then information is directly written to this object (note this prevents the use of temporary files).
- **dpath** (*Optional[PathLike]*) – where to download the file. If unspecified *appname* is used to determine this. Mutually exclusive with fpath.
- **fname** (*Optional[str]*) – What to name the downloaded file. Defaults to the url basename. Mutually exclusive with fpath.
- **hash_prefix** (*None | str*) – If specified, download will retry / error if the file hash does not match this value. Defaults to None.
- **hasher** (*str | Hasher*) – If hash_prefix is specified, this indicates the hashing algorithm to apply to the file. Defaults to sha512.
- **chunksize** (*int, default=2 * 13**) – Download chunksize.
- **verbose** (*int, default=1*) – Verbosity level 0 or 1.

- **timeout** (*float, default=NoParam*) – Specify timeout in seconds for `urllib.request.urlopen()`. (if not specified, the global default timeout setting will be used) This only works for HTTP, HTTPS and FTP connections for blocking operations like the connection attempt.
- **progkw** (*Dict | None*) – if specified provides extra arguments to the progress iterator. See *`ubelt.progiter.ProgIter`* for available options.

Returns fpath - path to the downloaded file.

Return type str | PathLike

Raises

- **URLError - if there is problem downloading the url** –
- **RuntimeError - if the hash does not match the hash_prefix** –

Note: Based largely on code in pytorch [TorchDL] with modifications influenced by other resources [Shichao_2012] [SO_15644964] [SO_16694907].

References

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> from ubelt.util_download import *  # NOQA
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> fpath = download(url)
>>> print(basename(fpath))
rqwaDag.png
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import ubelt as ub
>>> import io
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> file = io.BytesIO()
>>> fpath = ub.download(url, file)
>>> file.seek(0)
>>> data = file.read()
>>> assert ub.hash_data(data, hasher='sha1').startswith('f79ea24571')
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> fpath = download(url, hasher='sha1', hash_prefix=
↪'f79ea24571da6ddd2ba12e3d57b515249ecb8a35')
Downloading url='http://i.imgur.com/rqwaDag.png' to fpath=...rqwaDag.png
...
...1233/1233... rate=... Hz, eta=..., total=...
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import pytest
>>> import ubelt as ub
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> #fpath = download(url, hasher='sha1', hash_prefix=
↪'f79ea24571da6ddd2ba12e3d57b515249ecb8a35')
>>> # test download from girder
>>> #url = 'https://data.kitware.com/api/v1/item/5b4039308d777f2e6225994c/download'
>>> #ub.download(url, hasher='sha512', hash_prefix='c98a46cb31205cf')
>>> with pytest.raises(RuntimeError):
>>>     ub.download(url, hasher='sha512', hash_prefix='BAD_HASH')
```

`ubelt.`**`dzip`**(*items1*, *items2*, *cls=<class 'dict'>*)

Zips elementwise pairs between items1 and items2 into a dictionary.

Values from items2 can be broadcast onto items1.

Parameters

- **items1** (*Iterable[KT]*) – full sequence
- **items2** (*Iterable[VT]*) – can either be a sequence of one item or a sequence of equal length to `items1`
- **cls** (*Type[dict], default=dict*) – dictionary type to use.

Returns similar to `dict(zip(items1, items2))`.

Return type Dict[KT, VT]

Example

```
>>> import ubelt as ub
>>> assert ub.dzip([1, 2, 3], [4]) == {1: 4, 2: 4, 3: 4}
>>> assert ub.dzip([1, 2, 3], [4, 4, 4]) == {1: 4, 2: 4, 3: 4}
>>> assert ub.dzip([], [4]) == {}
```

`ubelt.`**`ensure_app_cache_dir`**(*appname*, **args*)

Calls *`get_app_cache_dir()`* but ensures the directory exists.

Parameters

- **appname** (*str*) – the name of the application

- ***args** – any other subdirectories may be specified

Returns the path to the ensured directory

Return type str

SeeAlso: *get_app_cache_dir()*

Example

```
>>> import ubelt as ub
>>> dpath = ub.ensure_app_cache_dir('ubelt')
>>> assert exists(dpath)
```

ubelt.**ensure_app_config_dir**(*appname*, **args*)

Calls *get_app_config_dir()* but ensures the directory exists.

Parameters

- **appname** (*str*) – the name of the application
- ***args** – any other subdirectories may be specified

Returns the path to the ensured directory

Return type str

SeeAlso: *get_app_config_dir()*

Example

```
>>> import ubelt as ub
>>> dpath = ub.ensure_app_config_dir('ubelt')
>>> assert exists(dpath)
```

ubelt.**ensure_app_data_dir**(*appname*, **args*)

Calls *get_app_data_dir()* but ensures the directory exists.

Parameters

- **appname** (*str*) – the name of the application
- ***args** – any other subdirectories may be specified

Returns the path to the ensured directory

Return type str

SeeAlso: *get_app_data_dir()*

Example

```
>>> import ubelt as ub
>>> dpath = ub.ensure_app_data_dir('ubelt')
>>> assert exists(dpath)
```

ubelt.**ensure_unicode**(*text*)

Casts bytes into utf8 (mostly for python2 compatibility)

Parameters **text** (*str | bytes*) – text to ensure is decoded as unicode

Returns str

References

[SO_12561063] http://stackoverflow.com/questions/12561063/extract-data-from-file

Example

```
>>> from ubelt.util_str import *
>>> import codecs  # NOQA
>>> assert ensure_unicode('my ünicôdé strıng') == 'my ünicôdé strıng'
>>> assert ensure_unicode('text1') == 'text1'
>>> assert ensure_unicode('text1'.encode('utf8')) == 'text1'
>>> assert ensure_unicode('ï»¿text1'.encode('utf8')) == 'ï»¿text1'
>>> assert (codecs.BOM_UTF8 + 'text»¿'.encode('utf8')).decode('utf8')
```

ubelt.**ensuredir**(*dpath*, *mode=1023*, *verbose=0*, *recreate=False*)

Ensures that directory will exist. Creates new dir with sticky bits by default

Parameters

- **dpath** (*str | PathLike | Tuple[str | PathLike]*) – dir to ensure. Can also be a tuple to send to join
- **mode** (*int*) – octal mode of directory
- **verbose** (*int*) – verbosity
- **recreate** (*bool*) – if True removes the directory and all of its contents and creates a fresh new directory. USE CAREFULLY.

Returns path - the ensured directory

Return type str

SeeAlso: *ubelt.Path.ensuredir()*

Note: This function is not thread-safe in Python2

Example

```
>>> from ubelt.util_path import *  # NOQA
>>> import ubelt as ub
>>> cache_dpath = ub.ensure_app_cache_dir('ubelt')
>>> dpath = join(cache_dpath, 'ensuredir')
>>> if exists(dpath):
...     os.rmdir(dpath)
>>> assert not exists(dpath)
>>> ub.ensuredir(dpath)
>>> assert exists(dpath)
>>> os.rmdir(dpath)
```

`ubelt.`**`expandpath`**(*path*)

Shell-like environment variable and tilde path expansion.

Parameters **path** (*str | PathLike*) – string representation of a path

Returns expanded path

Return type str

Example

```
>>> from ubelt.util_path import *  # NOQA
>>> import ubelt as ub
>>> assert normpath(ub.expandpath('~/foo')) == join(ub.userhome(), 'foo')
>>> assert ub.expandpath('foo') == 'foo'
```

`ubelt.`**`find_duplicates`**(*items*, *k=2*, *key=None*)

Find all duplicate items in a list.

Search for all items that appear more than `k` times and return a mapping from each (k)-duplicate item to the positions it appeared in.

Parameters

- **items** (*Iterable[T]*) – hashable items possibly containing duplicates
- **k** (*int, default=2*) – only return items that appear at least `k` times.
- **key** (*Callable[[T], Any], default=None*) – Returns indices where *key(items[i])* maps to a particular value at least k times.

Returns maps each duplicate item to the indices at which it appears

Return type dict[T, List[int]]

Notes

Similar to `more_itertools.duplicates_everseen()`, `more_itertools.duplicates_justseen()`.

Example

```
>>> import ubelt as ub
>>> items = [0, 0, 1, 2, 3, 3, 0, 12, 2, 9]
>>> duplicates = ub.find_duplicates(items)
>>> # Duplicates are a mapping from each item that occurs 2 or more
>>> # times to the indices at which they occur.
>>> assert duplicates == {0: [0, 1, 6], 2: [3, 8], 3: [4, 5]}
>>> # You can set k=3 if you want to don't mind duplicates but you
>>> # want to find triplicates or quadruplets etc.
>>> assert ub.find_duplicates(items, k=3) == {0: [0, 1, 6]}
```

Example

```
>>> import ubelt as ub
>>> items = [0, 0, 1, 2, 3, 3, 0, 12, 2, 9]
>>> # note: k can less then 2
>>> duplicates = ub.find_duplicates(items, k=0)
>>> print(ub.repr2(duplicates, nl=0))
{0: [0, 1, 6], 1: [2], 2: [3, 8], 3: [4, 5], 9: [9], 12: [7]}
```

Example

```
>>> import ubelt as ub
>>> items = [10, 11, 12, 13, 14, 15, 16]
>>> duplicates = ub.find_duplicates(items, key=lambda x: x // 2)
>>> print(ub.repr2(duplicates, nl=0))
{5: [0, 1], 6: [2, 3], 7: [4, 5]}
```

`ubelt.find_exe`(*name*, *multi=False*, *path=None*)

Locate a command.

Search your local filesystem for an executable and return the first matching file with executable permission.

Parameters

- **name** (*str | PathLike*) – globstr of matching filename
- **multi** (*bool, default=False*) – if True return all matches instead of just the first.
- **path** (*str | PathLike | Iterable[str | PathLike] | None, default=None*) – overrides the system PATH variable.

Returns returns matching executable(s).

Return type str | List[str] | None

SeeAlso: `shutil.which()` - which is available in Python 3.3+.

Note: This is essentially the `which` UNIX command

References

Example

```
>>> find_exe('ls')
>>> find_exe('ping')
>>> assert find_exe('which') == find_exe(find_exe('which'))
>>> find_exe('which', multi=True)
>>> find_exe('ping', multi=True)
>>> find_exe('cmake', multi=True)
>>> find_exe('nvcc', multi=True)
>>> find_exe('noexist', multi=True)
```

Example

```
>>> import ubelt as ub
>>> assert not ub.find_exe('noexist', multi=False)
>>> assert ub.find_exe('ping', multi=False) or ub.find_exe('ls', multi=False)
>>> assert not ub.find_exe('noexist', multi=True)
>>> assert ub.find_exe('ping', multi=True) or ub.find_exe('ls', multi=True)
```

Benchmark:

```
>>> # xdoctest: +IGNORE_WANT
>>> import ubelt as ub
>>> import shutil
>>> for timer in ub.Timerit(100, bestof=10, label='ub.find_exe'):
>>>     ub.find_exe('which')
>>> for timer in ub.Timerit(100, bestof=10, label='shutil.which'):
>>>     shutil.which('which')
Timed best=58.71 µs, mean=59.64 ± 0.96 µs for ub.find_exe
Timed best=72.75 µs, mean=73.07 ± 0.22 µs for shutil.which
```

`ubelt.find_path`(*name*, *path=None*, *exact=False*)

Search for a file or directory on your local filesystem by name (file must be in a directory specified in a PATH environment variable)

Parameters

- **name** (*str | PathLike*) – file name to match. If exact is False this may be a glob pattern
- **path** (*str | Iterable[str | PathLike], default=None*) – list of directories to search either specified as an `os.pathsep` separated string or a list of directories. Defaults to environment PATH.
- **exact** (*bool, default=False*) – if True, only returns exact matches.

Yields *str* – candidate - a path that matches `name`

Note: Running with `name=''` (i.e. `ub.find_path('')`) will simply yield all directories in your PATH.

Note: For recursive behavior set `path=(d for d, _, _ in os.walk('.'))`, where '.' might be replaced by the root directory of interest.

Example

```
>>> list(find_path('ping', exact=True))
>>> list(find_path('bin'))
>>> list(find_path('*cc*'))
>>> list(find_path('cmake*'))
```

Example

```
>>> import ubelt as ub
>>> from os.path import dirname
>>> path = dirname(dirname(ub.util_platform.__file__))
>>> res = sorted(find_path('ubelt/util_*.py', path=path))
>>> assert len(res) >= 10
>>> res = sorted(find_path('ubelt/util_platform.py', path=path, exact=True))
>>> print(res)
>>> assert len(res) == 1
```

`ubelt.`**`flatten`**(*nested*)

Transforms a nested iterable into a flat iterable.

Parameters **nested** (*Iterable[Iterable[Any]]*) – list of lists

Returns flattened items

Return type Iterable[Any]

Notes

Equivalent to `more_itertools.flatten()` and `itertools.chain.from_iterable()`.

Example

```
>>> import ubelt as ub
>>> nested = [['a', 'b'], ['c', 'd']]
>>> list(ub.flatten(nested))
['a', 'b', 'c', 'd']
```

`ubelt.`**`get_app_cache_dir`**(*appname*, **args*)

Returns a writable directory for an application. This should be used for temporary deletable data.

Parameters

- **appname** (*str*) – the name of the application
- ***args** – any other subdirectories may be specified

Returns the path to the ensured directory

Return type str

Returns dpath - writable cache directory for this application

Return type str

SeeAlso: *ensure_app_cache_dir()*

ubelt.**get_app_config_dir**(*appname*, **args*)

Returns a writable directory for an application This should be used for persistent configuration files.

Parameters

- **appname** (*str*) – the name of the application
- ***args** – any other subdirectories may be specified

Returns dpath - writable config directory for this application

Return type str

SeeAlso: *ensure_app_config_dir()*

ubelt.**get_app_data_dir**(*appname*, **args*)

Returns a writable directory for an application. This should be used for temporary deletable data.

Parameters

- **appname** (*str*) – the name of the application
- ***args** – any other subdirectories may be specified

Returns dpath - writable data directory for this application

Return type str

SeeAlso: *ensure_app_data_dir()*

ubelt.**grabdata**(*url*, *fpath=None*, *dpath=None*, *fname=None*, *redo=False*, *verbose=1*, *appname=None*, *hash_prefix=None*, *hasher='sha512'*, ***download_kw*)

Downloads a file, caches it, and returns its local path.

If unspecified the location and name of the file is chosen automatically. A hash_prefix can be specified to verify the integrity of the downloaded data.

Parameters

- **url** (*str*) – url of the file to download
- **fpath** (*Optional[str | PathLike]*) – The full path to download the file to. If unspecified, the arguments *dpath* and *fname* are used to determine this.
- **dpath** (*Optional[str | PathLike]*) – where to download the file. If unspecified *appname* is used to determine this. Mutually exclusive with fpath.
- **fname** (*Optional[str]*) – What to name the downloaded file. Defaults to the url basename. Mutually exclusive with fpath.

- **redo** (*bool, default=False*) – if True forces redownload of the file
- **verbose** (*bool, default=True*) – verbosity flag
- **appname** (*str*) – set dpath to *ub.get_app_cache_dir(appname)*. Mutually exclusive with dpath and fpath.
- **hash_prefix** (*None | str*) – If specified, grabdata verifies that this matches the hash of the file, and then saves the hash in a adjacent file to certify that the download was successful. Defaults to None.
- **hasher** (*str | Hasher*) – If hash_prefix is specified, this indicates the hashing algorithm to apply to the file. Defaults to sha512. NOTE: Only pass hasher as a string. Passing as an instance is deprecated and can cause unexpected results.
- ****download_kw** – additional kwargs to pass to `ubelt.util_download.download()`

Returns fpath - path to downloaded or cached file.

Return type str | PathLike

CommandLine

```
xdoctest -m ubelt.util_download grabdata --network
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import ubelt as ub
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> fpath = ub.grabdata(url, fname='mario.png')
>>> result = basename(fpath)
>>> print(result)
mario.png
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import ubelt as ub
>>> import json
>>> fname = 'foo.bar'
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> prefix1 = '944389a39dfb8fa9'
>>> fpath = ub.grabdata(url, fname=fname, hash_prefix=prefix1, verbose=3)
>>> stamp_fpath = ub.Path(fpath + '.stamp_sha512.json')
>>> assert json.loads(stamp_fpath.read_text())['hash'][0].startswith(prefix1)
>>> # Check that the download doesn't happen again
>>> fpath = ub.grabdata(url, fname=fname, hash_prefix=prefix1)
>>> # todo: check file timestamps have not changed
>>> #
>>> # Check redo works with hash
>>> fpath = ub.grabdata(url, fname=fname, hash_prefix=prefix1, redo=True)
>>> # todo: check file timestamps have changed
```

(continues on next page)

(continued from previous page)

```
>>> #
>>> # Check that a redownload occurs when the stamp is changed
>>> with open(stamp_fpath, 'w') as file:
>>>     file.write('corrupt-stamp')
>>> fpath = ub.grabdata(url, fname=fname, hash_prefix=prefix1)
>>> assert json.loads(stamp_fpath.read_text())['hash'][0].startswith(prefix1)
>>> #
>>> # Check that a redownload occurs when the stamp is removed
>>> ub.delete(stamp_fpath)
>>> with open(fpath, 'w') as file:
>>>     file.write('corrupt-data')
>>> assert not ub.hash_file(fpath, base='hex', hasher='sha512').startswith(prefix1)
>>> fpath = ub.grabdata(url, fname=fname, hash_prefix=prefix1)
>>> assert ub.hash_file(fpath, base='hex', hasher='sha512').startswith(prefix1)
>>> #
>>> # Check that requesting new data causes redownload
>>> #url2 = 'https://data.kitware.com/api/v1/item/5b4039308d777f2e6225994c/download'
>>> #prefix2 = 'c98a46cb31205cf'  # hack SSL
>>> url2 = 'http://i.imgur.com/rqwaDag.png'
>>> prefix2 = '944389a39dfb8fa9'
>>> fpath = ub.grabdata(url2, fname=fname, hash_prefix=prefix2)
>>> assert json.loads(stamp_fpath.read_text())['hash'][0].startswith(prefix2)
```

`ubelt.`**`group_items`**(*items*, *key*)

Groups a list of items by group id.

Parameters

- **items** (*Iterable[VT]*) – a list of items to group
- **key** (*Iterable[KT] | Callable[[VT], KT]*) – either a corresponding list of group-ids for each item or a function used to map each item to a group-id.

Returns a mapping from each group id to the list of corresponding items

Return type dict[KT, List[VT]]

Example

```
>>> import ubelt as ub
>>> items    = ['ham',     'jam',   'spam',     'eggs',    'cheese', 'banana']
>>> groupids = ['protein', 'fruit', 'protein',  'protein', 'dairy',  'fruit']
>>> id_to_items = ub.group_items(items, groupids)
>>> print(ub.repr2(id_to_items, nl=0))
{'dairy': ['cheese'], 'fruit': ['jam', 'banana'], 'protein': ['ham', 'spam', 'eggs
↪']}
```

`ubelt.`**`hash_data`**(*data*, *hasher=NoParam*, *base=NoParam*, *types=False*, *convert=False*, *extensions=None*)

Get a unique hash depending on the state of the data.

Parameters

- **data** (*object*) – Any sort of loosely organized data

- **hasher** (*str | Hasher, default='sha512'*) – string code or a hash algorithm from hashlib. Valid hashing algorithms are defined by `hashlib.algorithms_guaranteed` (e.g. 'sha1', 'sha512', 'md5') as well as 'xxh32' and 'xxh64' if `xxhash` is installed.
- **base** (*List[str] | str, default='hex'*) – list of symbols or shorthand key. Valid keys are 'abc', 'hex', and 'dec'.
- **types** (*bool*) – If True data types are included in the hash, otherwise only the raw data is hashed. Defaults to False.
- **convert** (*bool, default=True*) – if True, try and convert the data to json an the json is hashed instead. This can improve runtime in some instances, however the hash may differ from the case where convert=False.
- **extensions** (*HashableExtensions*) – a custom `HashableExtensions` instance that can overwrite or define how different types of objects are hashed.

Note: The types allowed are specified by the HashableExtensions object. By default ubelt will register:

OrderedDict, uuid.UUID, np.random.RandomState, np.int64, np.int32, np.int16, np.int8, np.uint64, np.uint32, np.uint16, np.uint8, np.float16, np.float32, np.float64, np.float128, np.ndarray, bytes, str, int, float, long (in python2), list, tuple, set, and dict

Returns text representing the hashed data

Return type str

Note: The alphabet26 base is a pretty nice base, I recommend it. However we default to `base='hex'` because it is standard. You can try the alphabet26 base by setting `base='abc'`.

Example

```
>>> import ubelt as ub
>>> print(ub.hash_data([1, 2, (3, '4')], convert=False))
60b758587f599663931057e6ebdf185a...
>>> print(ub.hash_data([1, 2, (3, '4')], base='abc',  hasher='sha512')[:32])
hsrgqvfiuxvvhcdnypivhhthmrolkzej
```

`ubelt.hash_file`(*fpath*, *blocksize=1048576*, *stride=1*, *maxbytes=None*, *hasher=NoParam*, *base=NoParam*)

Hashes the data in a file on disk.

The results of this function agree with the standard UNIX commands (e.g. sha1sum, sha512sum, md5sum, etc...)

Parameters

- **fpath** (*PathLike*) – location of the file to be hashed.
- **blocksize** (*int, default=2 * 20**) – Amount of data to read and hash at a time. There is a trade off and the optimal number will depend on specific hardware. This number was chosen to be optimal on a developer system. See "dev/bench_hash_file" for methodology to choose this number for your use case.
- **stride** (*int, default=1*) – strides > 1 skip data to hash, useful for faster hashing, but less accurate, also makes hash dependent on blocksize.

- **maxbytes** (*int | None*) – if specified, only hash the leading *maxbytes* of data in the file.
- **hasher** (*str | Hasher, default='sha512'*) – string code or a hash algorithm from hashlib. Valid hashing algorithms are defined by `hashlib.algorithms_guaranteed` (e.g. 'sha1', 'sha512', 'md5') as well as 'xxh32' and 'xxh64' if `xxhash` is installed.

  TODO: add logic such that you can update an existing hasher
- **base** (*List[str] | str, default='hex'*) – list of symbols or shorthand key. Valid keys are 'abc', 'hex', and 'dec'.

Note: For better hashes keep stride = 1. For faster hashes set stride > 1. Blocksize matters when stride > 1.

References

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> dpath = ub.Path.appdir('ubelt/tests/test-hash').ensuredir()
>>> fpath = dpath / 'tmp1.txt'
>>> fpath.write_text('foobar')
>>> print(ub.hash_file(fpath, hasher='sha1', base='hex'))
8843d7f92416211de9ebb963ff4ce28125932878
```

Example

```
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('ubelt/tests/test-hash').ensuredir()
>>> fpath = dpath / 'tmp2.txt'
>>> # We have the ability to only hash at most ``maxbytes`` in a file
>>> fpath.write_text('abcdefghijklmnop')
>>> h0 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=11, blocksize=3)
>>> h1 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=32, blocksize=3)
>>> h2 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=32, blocksize=32)
>>> h3 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=16, blocksize=1)
>>> h4 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=16, blocksize=18)
>>> assert h1 == h2 == h3 == h4
>>> assert h1 != h0
```

```
>>> # Using a stride makes the result dependent on the blocksize
>>> h0 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=11, blocksize=3,␣
↪stride=2)
>>> h1 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=32, blocksize=3,␣
↪stride=2)
>>> h2 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=32, blocksize=32,␣
↪stride=2)
>>> h3 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=16, blocksize=1,␣
↪stride=2)
>>> h4 = ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=16, blocksize=18,␣
↪stride=2)
```

(continues on next page)

(continued from previous page)

```
>>> assert h1 != h2 != h3
>>> assert h1 == h0
>>> assert h2 == h4
```

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> dpath = ub.ensure_app_cache_dir('ubelt/tests/test-hash')
>>> fpath = ub.touch(join(dpath, 'empty_file'))
>>> # Test that the output is the same as sha1sum executable
>>> if ub.find_exe('sha1sum'):
>>>     want = ub.cmd(['sha1sum', fpath], verbose=2)['out'].split(' ')[0]
>>>     got = ub.hash_file(fpath, hasher='sha1')
>>>     print('want = {!r}'.format(want))
>>>     print('got = {!r}'.format(got))
>>>     assert want.endswith(got)
>>> # Do the same for sha512 sum and md5sum
>>> if ub.find_exe('sha512sum'):
>>>     want = ub.cmd(['sha512sum', fpath], verbose=2)['out'].split(' ')[0]
>>>     got = ub.hash_file(fpath, hasher='sha512')
>>>     print('want = {!r}'.format(want))
>>>     print('got = {!r}'.format(got))
>>>     assert want.endswith(got)
>>> if ub.find_exe('md5sum'):
>>>     want = ub.cmd(['md5sum', fpath], verbose=2)['out'].split(' ')[0]
>>>     got = ub.hash_file(fpath, hasher='md5')
>>>     print('want = {!r}'.format(want))
>>>     print('got = {!r}'.format(got))
>>>     assert want.endswith(got)
```

`ubelt.highlight_code`(*text*, *lexer_name='python'*, ***kwargs*)

Highlights a block of text using ANSI tags based on language syntax.

Parameters

- **text** (*str*) – plain text to highlight
- **lexer_name** (*str*) – name of language. eg: python, docker, c++
- ****kwargs** – passed to pygments.lexers.get_lexer_by_name

Returns text - highlighted text If pygments is not installed, the plain text is returned.

Return type str

Example

```
>>> import ubelt as ub
>>> text = 'import ubelt as ub; print(ub)'
>>> new_text = ub.highlight_code(text)
>>> print(new_text)
```

ubelt.**hzcat**(*args*, *sep=''*)

Horizontally concatenates strings preserving indentation

Concatenates a list of objects ensuring that the next item in the list is all the way to the right of any previous items.

Parameters

- **args** (*List[str]*) – strings to concatenate
- **sep** (*str, default=''*) – separator

Example1:

```
>>> import ubelt as ub
>>> B = ub.repr2([[1, 2], [3, 457]], nl=1, cbr=True, trailsep=False)
>>> C = ub.repr2([[5, 6], [7, 8]], nl=1, cbr=True, trailsep=False)
>>> args = ['A = ', B, ' * ', C]
>>> print(ub.hzcat(args))
A = [[1, 2],   * [[5, 6],
     [3, 457]]    [7, 8]]
```

Example2:

```
>>> import ubelt as ub
>>> import unicodedata
>>> aa = unicodedata.normalize('NFD', 'á')  # a unicode char with len2
>>> B = ub.repr2([['', aa], [aa, aa, aa]], nl=1, si=True, cbr=True,␣
↪trailsep=False)
>>> C = ub.repr2([[5, 6], [7, '']], nl=1, si=True, cbr=True, trailsep=False)
>>> args = ['A', '=', B, '*', C]
>>> print(ub.hzcat(args, sep=''))
A=[[, á],   *[[5, 6],
   [á, á, á]]  [7, ]]
```

ubelt.**identity**(*arg=None*, **args*, ***kwargs*)

The identity function. Simply returns the value of its first input.

All other inputs are ignored. Defaults to None if called without args.

Parameters

- **arg** (*object, default=None*) – some value
- ***args** – ignored
- ****kwargs** – ignored

Returns arg - the same value

Return type object

Example

```
>>> import ubelt as ub
>>> ub.identity(42)
42
>>> ub.identity(42, 42)
42
>>> ub.identity()
None
```

ubelt.**import_module_from_name**(*modname*)

Imports a module from its string name (i.e. `__name__`)

Parameters **modname** (*str*) – module name

Returns module

Return type ModuleType

SeeAlso: *import_module_from_path()*

Example

```
>>> # test with modules that won't be imported in normal circumstances
>>> # todo write a test where we guarantee this
>>> modname_list = [
>>>     'pickletools',
>>>     'lib2to3.fixes.fix_apply',
>>> ]
>>> #assert not any(m in sys.modules for m in modname_list)
>>> modules = [import_module_from_name(modname) for modname in modname_list]
>>> assert [m.__name__ for m in modules] == modname_list
>>> assert all(m in sys.modules for m in modname_list)
```

ubelt.**import_module_from_path**(*modpath*, *index=- 1*)

Imports a module via its path

Parameters

- **modpath** (*str | PathLike*) – Path to the module on disk or within a zipfile.
- **index** (*int*) – Location at which we modify PYTHONPATH if necessary. If your module name does not conflict, the safest value is -1, However, if there is a conflict, then use an index of 0. The default may change to 0 in the future.

Returns the imported module

Return type ModuleType

References

Note: If the module is part of a package, the package will be imported first. These modules may cause problems when reloading via IPython magic

This can import a module from within a zipfile. To do this modpath should specify the path to the zipfile and the path to the module within that zipfile separated by a colon or pathsep. E.g. "/path/to/archive.zip:mymodule.pl"

Warning: It is best to use this with paths that will not conflict with previously existing modules.

If the modpath conflicts with a previously existing module name. And the target module does imports of its own relative to this conflicting path. In this case, the module that was loaded first will win.

For example if you try to import '/foo/bar/pkg/mod.py' from the folder structure:

```
- foo/
  +- bar/
     +- pkg/
        +  __init__.py
        |- mod.py
        |- helper.py
```

If there exists another module named `pkg` already in sys.modules and mod.py does something like `from . import helper`, Python will assume helper belongs to the `pkg` module already in sys.modules. This can cause a NameError or worse — a incorrect helper module.

SeeAlso: *import_module_from_name()*

Example

```
>>> import xdoctest
>>> modpath = xdoctest.__file__
>>> module = import_module_from_path(modpath)
>>> assert module is xdoctest
```

Example

```
>>> # Test importing a module from within a zipfile
>>> import zipfile
>>> from xdoctest import utils
>>> from os.path import join, expanduser, normpath
>>> dpath = expanduser('~/.cache/xdoctest')
>>> dpath = utils.ensuredir(dpath)
>>> #dpath = utils.TempDir().ensure()
>>> # Write to an external module named bar
>>> external_modpath = join(dpath, 'bar.py')
>>> # For pypy support we have to write this using with
>>> with open(external_modpath, 'w') as file:
>>>     file.write('testvar = 1')
```

(continues on next page)

(continued from previous page)

```
>>> internal = 'folder/bar.py'
>>> # Move the external bar module into a zipfile
>>> zippath = join(dpath, 'myzip.zip')
>>> with zipfile.ZipFile(zippath, 'w') as myzip:
>>>     myzip.write(external_modpath, internal)
>>> # Import the bar module from within the zipfile
>>> modpath = zippath + ':' + internal
>>> modpath = zippath + os.path.sep + internal
>>> module = import_module_from_path(modpath)
>>> assert normpath(module.__name__) == normpath('folder/bar')
>>> assert module.testvar == 1
```

Example

```
>>> import pytest
>>> with pytest.raises(IOError):
>>>     import_module_from_path('does-not-exist')
>>> with pytest.raises(IOError):
>>>     import_module_from_path('does-not-exist.zip/')
```

`ubelt.`**`indent`**(*text*, *prefix=' '*)

Indents a block of text

Parameters

- **text** (*str*) – text to indent
- **prefix** (*str, default = ' '*) – prefix to add to each line

Returns indented text

Return type str

Example

```
>>> import ubelt as ub
>>> NL = chr(10)  # newline character
>>> text = 'Lorem ipsum' + NL + 'dolor sit amet'
>>> prefix = '    '
>>> result = ub.indent(text, prefix)
>>> assert all(t.startswith(prefix) for t in result.split(NL))
```

`ubelt.`**`indexable_allclose`**(*dct1*, *dct2*, *rel_tol=1e-09*, *abs_tol=0.0*, *return_info=False*)

Walks through two nested data structures and ensures that everything is roughly the same.

Parameters

- **dct1** (*dict*) – a nested indexable item
- **dct2** (*dict*) – a nested indexable item
- **rel_tol** (*float*) – maximum difference for being considered "close", relative to the magnitude of the input values

- **abs_tol** (*float*) – maximum difference for being considered "close", regardless of the magnitude of the input values
- **return_info** (*bool, default=False*) – if true, return extra info

Returns A boolean result if `return_info` is false, otherwise a tuple of the boolean result and an "info" dict containing detailed results indicating what matched and what did not.

Return type bool | Tuple[bool, Dict]

Example

```
>>> import ubelt as ub
>>> dct1 = {
>>>     'foo': [1.222222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> dct2 = {
>>>     'foo': [1.22222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> flag, return_info =  ub.indexable_allclose(dct1, dct2, return_info=True)
>>> print('return_info = {}'.format(ub.repr2(return_info, nl=1)))
>>> print('flag = {!r}'.format(flag))
```

```
>>> walker1 = return_info['walker1']
>>> for p1, v1, v2  in return_info['faillist']:
>>>     v1_ = walker1[p1]
>>>     print('*fail p1, v1, v2 = {}, {}, {}'.format(p1, v1, v2))
>>> for p1 in return_info['passlist']:
>>>     v1_ = walker1[p1]
>>>     print('*pass p1, v1_ = {}, {}'.format(p1, v1_))
>>> assert not flag
```

```
>>> import ubelt as ub
>>> dct1 = {
>>>     'foo': [1.000000000000000000000001, 1.],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> dct2 = {
>>>     'foo': [0.9999999999999999, 1.],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> flag, return_info =  ub.indexable_allclose(dct1, dct2, return_info=True)
>>> print('return_info = {}'.format(ub.repr2(return_info, nl=1)))
>>> print('flag = {!r}'.format(flag))
```

Example

```
>>> import ubelt as ub
>>> flag, return_info =  ub.indexable_allclose([], [], return_info=True)
>>> print('return_info = {!r}'.format(return_info))
>>> print('flag = {!r}'.format(flag))
```

Example

```
>>> import ubelt as ub
>>> flag =  ub.indexable_allclose([], [], return_info=False)
>>> print('flag = {!r}'.format(flag))
```

Example

```
>>> import ubelt as ub
>>> flag, return_info =  ub.indexable_allclose([], [1], return_info=True)
>>> print('return_info = {!r}'.format(return_info))
>>> print('flag = {!r}'.format(flag))
```

`ubelt.`**`inject_method`**(*self*, *func*, *name=None*)

Injects a function into an object instance as a bound method

The main use case of this function is for monkey patching. While monkey patching is sometimes necessary it should generally be avoided. Thus, we simply remind the developer that there might be a better way.

Parameters

- **self** (*T*) – Instance to inject a function into.
- **func** (*Callable[..., Any]*) – The function to inject (must contain an arg for self).
- **name** (*str, default=None*) – Name of the method. optional. If not specified the name of the function is used.

Example

```
>>> import ubelt as ub
>>> class Foo(object):
>>>     def bar(self):
>>>         return 'bar'
>>> def baz(self):
>>>     return 'baz'
>>> self = Foo()
>>> assert self.bar() == 'bar'
>>> assert not hasattr(self, 'baz')
>>> ub.inject_method(self, baz)
>>> assert not hasattr(Foo, 'baz'), 'should only change one instance'
>>> assert self.baz() == 'baz'
>>> ub.inject_method(self, baz, 'bar')
>>> assert self.bar() == 'baz'
```

`ubelt.invert_dict(dict_, unique_vals=True)`

Swaps the keys and values in a dictionary.

Parameters

- **dict_** (*Dict[KT, VT]*) – dictionary to invert
- **unique_vals** (*bool, default=True*) – if False, the values of the new dictionary are sets of the original keys.

Returns the inverted dictionary

Return type Dict[VT, KT] | Dict[VT, Set[KT]]

Note: The must values be hashable.

If the original dictionary contains duplicate values, then only one of the corresponding keys will be returned and the others will be discarded. This can be prevented by setting `unique_vals=False`, causing the inverted keys to be returned in a set.

Example

```
>>> import ubelt as ub
>>> dict_ = {'a': 1, 'b': 2}
>>> inverted = ub.invert_dict(dict_)
>>> assert inverted == {1: 'a', 2: 'b'}
```

Example

```
>>> import ubelt as ub
>>> dict_ = ub.odict([(2, 'a'), (1, 'b'), (0, 'c'), (None, 'd')])
>>> inverted = ub.invert_dict(dict_)
>>> assert list(inverted.keys())[0] == 'a'
```

Example

```
>>> import ubelt as ub
>>> dict_ = {'a': 1, 'b': 0, 'c': 0, 'd': 0, 'f': 2}
>>> inverted = ub.invert_dict(dict_, unique_vals=False)
>>> assert inverted == {0: {'b', 'c', 'd'}, 1: {'a'}, 2: {'f'}}
```

`ubelt.iter_window(iterable, size=2, step=1, wrap=False)`

Iterates through iterable with a window size. This is essentially a 1D sliding window.

Parameters

- **iterable** (*Iterable[T]*) – an iterable sequence
- **size** (*int, default=2*) – sliding window size
- **step** (*int, default=1*) – sliding step size
- **wrap** (*bool, default=False*) – wraparound flag

Returns returns a possibly overlapping windows in a sequence

Return type Iterable[T]

Notes

Similar to `more_itertools.windowed()`, Similar to `more_itertools.pairwise()`, Similar to `more_itertools.triplewise()`, Similar to `more_itertools.sliding_window()`

Example

```
>>> import ubelt as ub
>>> iterable = [1, 2, 3, 4, 5, 6]
>>> size, step, wrap = 3, 1, True
>>> window_iter = ub.iter_window(iterable, size, step, wrap)
>>> window_list = list(window_iter)
>>> print('window_list = %r' % (window_list,))
window_list = [(1, 2, 3), (2, 3, 4), (3, 4, 5), (4, 5, 6), (5, 6, 1), (6, 1, 2)]
```

Example

```
>>> import ubelt as ub
>>> iterable = [1, 2, 3, 4, 5, 6]
>>> size, step, wrap = 3, 2, True
>>> window_iter = ub.iter_window(iterable, size, step, wrap)
>>> window_list = list(window_iter)
>>> print('window_list = {!r}'.format(window_list))
window_list = [(1, 2, 3), (3, 4, 5), (5, 6, 1)]
```

Example

```
>>> import ubelt as ub
>>> iterable = [1, 2, 3, 4, 5, 6]
>>> size, step, wrap = 3, 2, False
>>> window_iter = ub.iter_window(iterable, size, step, wrap)
>>> window_list = list(window_iter)
>>> print('window_list = {!r}'.format(window_list))
window_list = [(1, 2, 3), (3, 4, 5)]
```

Example

```
>>> import ubelt as ub
>>> iterable = []
>>> size, step, wrap = 3, 2, False
>>> window_iter = ub.iter_window(iterable, size, step, wrap)
>>> window_list = list(window_iter)
>>> print('window_list = {!r}'.format(window_list))
window_list = []
```

`ubelt.iterable(obj, strok=False)`

Checks if the input implements the iterator interface. An exception is made for strings, which return False unless `strok` is True

Parameters

- **obj** (*object*) – a scalar or iterable input
- **strok** (*bool, default=False*) – if True allow strings to be interpreted as iterable

Returns True if the input is iterable

Return type bool

Example

```
>>> import ubelt as ub
>>> obj_list = [3, [3], '3', (3,), [3, 4, 5], {}]
>>> result = [ub.iterable(obj) for obj in obj_list]
>>> assert result == [False, True, False, True, True, True]
>>> result = [ub.iterable(obj, strok=True) for obj in obj_list]
>>> assert result == [False, True, True, True, True, True]
```

`ubelt.map_keys(func, dict_)`

Apply a function to every key in a dictionary.

Creates a new dictionary with the same values and modified keys. An error is raised if the new keys are not unique.

Parameters

- **func** (*Callable[[KT], T] | Mapping[KT, T]*) – a function or indexable object
- **dict_** (*Dict[KT, VT]*) – a dictionary

Returns transformed dictionary

Return type Dict[T, VT]

Raises `Exception` – if multiple keys map to the same value

Example

```
>>> import ubelt as ub
>>> dict_ = {'a': [1, 2, 3], 'b': []}
>>> func = ord
>>> newdict = ub.map_keys(func, dict_)
>>> print(newdict)
>>> assert newdict == {97: [1, 2, 3], 98: []}
>>> dict_ = {0: [1, 2, 3], 1: []}
>>> func = ['a', 'b']
>>> newdict = ub.map_keys(func, dict_)
>>> print(newdict)
>>> assert newdict == {'a': [1, 2, 3], 'b': []}
```

`ubelt.map_vals(func, dict_)`

Apply a function to every value in a dictionary.

Creates a new dictionary with the same keys and modified values.

Parameters

- **func** (*Callable[[VT], T] | Mapping[VT, T]*) – a function or indexable object
- **dict_** (*Dict[KT, VT]*) – a dictionary

Returns transformed dictionary

Return type Dict[KT, T]

Example

```
>>> import ubelt as ub
>>> dict_ = {'a': [1, 2, 3], 'b': []}
>>> newdict = ub.map_vals(len, dict_)
>>> assert newdict ==  {'a': 3, 'b': 0}
```

Example

```
>>> # Can also use an indexable as ``func``
>>> import ubelt as ub
>>> dict_ = {'a': 0, 'b': 1}
>>> func = [42, 21]
>>> newdict = ub.map_vals(func, dict_)
>>> assert newdict ==  {'a': 42, 'b': 21}
>>> print(newdict)
```

`ubelt.memoize(func)`

memoization decorator that respects args and kwargs

In Python 3.9. The `functools` introduces the *cache* method, which is currently faster than memoize for simple functions [FunctoolsCache]. However, memoize can handle more general non-natively hashable inputs.

Parameters **func** (*Callable*) – live python function

Returns memoized wrapper

Return type Callable

References

Example

```
>>> import ubelt as ub
>>> closure = {'a': 'b', 'c': 'd'}
>>> incr = [0]
>>> def foo(key):
>>>     value = closure[key]
>>>     incr[0] += 1
```

(continues on next page)

(continued from previous page)

```
>>>     return value
>>> foo_memo = ub.memoize(foo)
>>> assert foo('a') == 'b' and foo('c') == 'd'
>>> assert incr[0] == 2
>>> print('Call memoized version')
>>> assert foo_memo('a') == 'b' and foo_memo('c') == 'd'
>>> assert incr[0] == 4
>>> assert foo_memo('a') == 'b' and foo_memo('c') == 'd'
>>> print('Counter should no longer increase')
>>> assert incr[0] == 4
>>> print('Closure changes result without memoization')
>>> closure = {'a': 0, 'c': 1}
>>> assert foo('a') == 0 and foo('c') == 1
>>> assert incr[0] == 6
>>> assert foo_memo('a') == 'b' and foo_memo('c') == 'd'
```

class ubelt.**memoize_method**(*func*)

Bases: object

memoization decorator for a method that respects args and kwargs

References

Example

```
>>> import ubelt as ub
>>> closure = {'a': 'b', 'c': 'd'}
>>> incr = [0]
>>> class Foo(object):
>>>     @memoize_method
>>>     def foo_memo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>>     def foo(self, key):
>>>         value = closure[key]
>>>         incr[0] += 1
>>>         return value
>>> self = Foo()
>>> assert self.foo('a') == 'b' and self.foo('c') == 'd'
>>> assert incr[0] == 2
>>> print('Call memoized version')
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> assert incr[0] == 4
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Counter should no longer increase')
>>> assert incr[0] == 4
>>> print('Closure changes result without memoization')
>>> closure = {'a': 0, 'c': 1}
>>> assert self.foo('a') == 0 and self.foo('c') == 1
>>> assert incr[0] == 6
```

(continues on next page)

(continued from previous page)

```
>>> assert self.foo_memo('a') == 'b' and self.foo_memo('c') == 'd'
>>> print('Constructing a new object should get a new cache')
>>> self2 = Foo()
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
>>> self2.foo_memo('a')
>>> assert incr[0] == 7
```

ubelt.**memoize_property**(*fget*)

Return a property attribute for new-style classes that only calls its getter on the first access. The result is stored and on subsequent accesses is returned, preventing the need to call the getter any more.

This decorator can either be used by itself or by decorating another property. In either case the method will always become a property.

Note: implementation is a modified version of [estebistec_memoize].

References

Example

```
>>> import ubelt as ub
>>> class C(object):
...     load_name_count = 0
...     @ub.memoize_property
...     def name(self):
...         "name's docstring"
...         self.load_name_count += 1
...         return "the name"
...     @ub.memoize_property
...     @property
...     def another_name(self):
...         "name's docstring"
...         self.load_name_count += 1
...         return "the name"
>>> c = C()
>>> c.load_name_count
0
>>> c.name
'the name'
>>> c.load_name_count
1
>>> c.name
'the name'
>>> c.load_name_count
1
>>> c.another_name
```

ubelt.**modname_to_modpath**(*modname*, *hide_init=True*, *hide_main=False*, *sys_path=None*)

Finds the path to a python module from its name.

Determines the path to a python module without directly import it

Converts the name of a module (__name__) to the path (__file__) where it is located without importing the module. Returns None if the module does not exist.

Parameters

- **modname** (*str*) – module filepath
- **hide_init** (*bool*) – if False, __init__.py will be returned for packages
- **hide_main** (*bool*) – if False, and hide_init is True, __main__.py will be returned for packages, if it exists.
- **sys_path** (*list, default=None*) – if specified overrides `sys.path`

Returns modpath - path to the module, or None if it doesn't exist

Return type str

Example

```
>>> modname = 'xdoctest.__main__'
>>> modpath = modname_to_modpath(modname, hide_main=False)
>>> assert modpath.endswith('__main__.py')
>>> modname = 'xdoctest'
>>> modpath = modname_to_modpath(modname, hide_init=False)
>>> assert modpath.endswith('__init__.py')
>>> # xdoctest: +REQUIRES(CPython)
>>> modpath = basename(modname_to_modpath('_ctypes'))
>>> assert 'ctypes' in modpath
```

`ubelt.`**`modpath_to_modname`**(*modpath*, *hide_init=True*, *hide_main=False*, *check=True*, *relativeto=None*)

Determines importable name from file path

Converts the path to a module (__file__) to the importable python name (__name__) without importing the module.

The filename is converted to a module name, and parent directories are recursively included until a directory without an __init__.py file is encountered.

Parameters

- **modpath** (*str*) – module filepath
- **hide_init** (*bool, default=True*) – removes the __init__ suffix
- **hide_main** (*bool, default=False*) – removes the __main__ suffix
- **check** (*bool, default=True*) – if False, does not raise an error if modpath is a dir and does not contain an __init__ file.
- **relativeto** (*str, default=None*) – if specified, all checks are ignored and this is considered the path to the root module.

Todo:

- **[ ] Does this need modification to support PEP 420?** https://www.python.org/dev/peps/pep-0420/

Returns modname

Return type str

Raises ValueError – if check is True and the path does not exist

Example

```
>>> from xdoctest import static_analysis
>>> modpath = static_analysis.__file__.replace('.pyc', '.py')
>>> modpath = modpath.replace('.pyc', '.py')
>>> modname = modpath_to_modname(modpath)
>>> assert modname == 'xdoctest.static_analysis'
```

Example

```
>>> import xdoctest
>>> assert modpath_to_modname(xdoctest.__file__.replace('.pyc', '.py')) == 'xdoctest
↪'
>>> assert modpath_to_modname(dirname(xdoctest.__file__.replace('.pyc', '.py'))) ==
↪'xdoctest'
```

Example

```
>>> # xdoctest: +REQUIRES(CPython)
>>> modpath = modname_to_modpath('_ctypes')
>>> modname = modpath_to_modname(modpath)
>>> assert modname == '_ctypes'
```

Example

```
>>> modpath = '/foo/libfoobar.linux-x86_64-3.6.so'
>>> modname = modpath_to_modname(modpath, check=False)
>>> assert modname == 'libfoobar'
```

ubelt.**named_product**(*_=None*, ***basis*)

Generates the Cartesian product of the basis.values(), where each generated item labeled by basis.keys().

In other words, given a dictionary that maps each "axes" (i.e. some variable) to its "basis" (i.e. the possible values that it can take), generate all possible points in that grid (i.e. unique assignments of variables to values).

Parameters

- _ (*Dict[str, List[VT]] | None, default=None*) – Use of this positional argument is not recommend. Instead specify all arguments as keyword args.

  If specified, this should be a dictionary is unioned with the keyword args. This exists to support ordered dictionaries before Python 3.6, and may eventually be removed.

- **basis** (*Dict[str, List[VT]]*) – A dictionary where the keys correspond to "columns" and the values are a list of possible values that "column" can take.

  I.E. each key corresponds to an "axes", the values are the list of possible values for that "axes".

Yields *Dict[str, VT]* – a "row" in the "longform" data containing a point in the Cartesian product.

Note: This function is similar to `itertools.product()`, the only difference is that the generated items are a dictionary that retains the input keys instead of an tuple.

This function used to be called "basis_product", but "named_product" might be more appropriate. This function exists in other places ([minstrel271_namedproduct], [pytb_namedproduct], and [Hettinger_namedproduct]).

References

Example

```
>>> # An example use case is looping over all possible settings in a
>>> # configuration dictionary for a grid search over parameters.
>>> import ubelt as ub
>>> basis = {
>>>     'arg1': [1, 2, 3],
>>>     'arg2': ['A1', 'B1'],
>>>     'arg3': [9999, 'Z2'],
>>>     'arg4': ['always'],
>>> }
>>> import ubelt as ub
>>> # sort input data for older python versions
>>> basis = ub.odict(sorted(basis.items()))
>>> got = list(ub.named_product(basis))
>>> print(ub.repr2(got, nl=-1))
[
    {'arg1': 1, 'arg2': 'A1', 'arg3': 9999, 'arg4': 'always'},
    {'arg1': 1, 'arg2': 'A1', 'arg3': 'Z2', 'arg4': 'always'},
    {'arg1': 1, 'arg2': 'B1', 'arg3': 9999, 'arg4': 'always'},
    {'arg1': 1, 'arg2': 'B1', 'arg3': 'Z2', 'arg4': 'always'},
    {'arg1': 2, 'arg2': 'A1', 'arg3': 9999, 'arg4': 'always'},
    {'arg1': 2, 'arg2': 'A1', 'arg3': 'Z2', 'arg4': 'always'},
    {'arg1': 2, 'arg2': 'B1', 'arg3': 9999, 'arg4': 'always'},
    {'arg1': 2, 'arg2': 'B1', 'arg3': 'Z2', 'arg4': 'always'},
    {'arg1': 3, 'arg2': 'A1', 'arg3': 9999, 'arg4': 'always'},
    {'arg1': 3, 'arg2': 'A1', 'arg3': 'Z2', 'arg4': 'always'},
    {'arg1': 3, 'arg2': 'B1', 'arg3': 9999, 'arg4': 'always'},
    {'arg1': 3, 'arg2': 'B1', 'arg3': 'Z2', 'arg4': 'always'}
]
```

Example

```
>>> import ubelt as ub
>>> list(ub.named_product(a=[1, 2, 3]))
[{'a': 1}, {'a': 2}, {'a': 3}]
>>> # xdoctest: +IGNORE_WANT
>>> list(ub.named_product(a=[1, 2, 3], b=[4, 5]))
[{'a': 1, 'b': 4},
 {'a': 1, 'b': 5},
```

(continues on next page)

(continued from previous page)

```
{'a': 2, 'b': 4},
{'a': 2, 'b': 5},
{'a': 3, 'b': 4},
{'a': 3, 'b': 5}]
```

ubelt.**odict**

alias of collections.OrderedDict

ubelt.**oset**

alias of *ubelt.orderedset.OrderedSet*

ubelt.**paragraph**(*text*)

Wraps multi-line strings and restructures the text to remove all newlines, heading, trailing, and double spaces.

Useful for writing log messages

Parameters **text** (*str*) – typically a multiline string

Returns the reduced text block

Return type str

Example

```
>>> import ubelt as ub
>>> text = (
>>>     '''
>>>     Lorem ipsum dolor sit amet, consectetur adipiscing
>>>     elit, sed do eiusmod tempor incididunt ut labore et
>>>     dolore magna aliqua.
>>>     ''')
>>> out = ub.paragraph(text)
>>> assert chr(10) in text
>>> assert chr(10) not in out
>>> print('text = {!r}'.format(text))
>>> print('out = {!r}'.format(out))
```

ubelt.**peek**(*iterable*, *default=NoParam*)

Look at the first item of an iterable. If the input is an iterator, then the next element is exhausted (i.e. a pop operation).

Parameters

- **iterable** (*Iterable[T]*) – an iterable
- **default** (*T*) – default item to return if the iterable is empty, otherwise a StopIteration error is raised

Returns

item - the first item of ordered sequence, a popped item from an iterator, or an arbitrary item from an unordered collection.

Return type T

Notes

Similar to `more_itertools.peekable()`

Example

```
>>> import ubelt as ub
>>> data = [0, 1, 2]
>>> ub.peek(data)
0
>>> iterator = iter(data)
>>> print(ub.peek(iterator))
0
>>> print(ub.peek(iterator))
1
>>> print(ub.peek(iterator))
2
>>> ub.peek(range(3))
0
>>> ub.peek([], 3)
3
```

`ubelt.platform_cache_dir()`

Returns a directory which should be writable for any application This should be used for temporary deletable data.

Returns path to the cache dir used by the current operating system

Return type str

`ubelt.platform_config_dir()`

Returns a directory which should be writable for any application This should be used for persistent configuration files.

Returns path to the cache dir used by the current operating system

Return type str

`ubelt.platform_data_dir()`

Returns path for user-specific data files

Returns path to the data dir used by the current operating system

Return type str

`ubelt.readfrom(fpath, aslines=False, errors='replace', verbose=None)`

Reads (utf8) text from a file.

Note: You probably should use `ub.Path(<fpath>).read_text()` instead. This function exists as a convenience for writing in Python2. After 2020-01-01, we may consider deprecating the function.

Parameters

- **fpath** (*str* | *PathLike*) – file path
- **aslines** (*bool*) – if True returns list of lines

- **verbose** (*bool*) – verbosity flag

Returns text from fpath (this is unicode)

Return type str

`ubelt.repr2(data, **kwargs)`

Makes a pretty string representation of `data`.

Makes a pretty and easy-to-doctest string representation. Has nice handling of common nested datatypes. This is an alternative to repr, and `pprint.pformat()`.

This output of this function are configurable. By default it aims to produce strings that are consistent, compact, and executable. This makes them great for doctests.

Note: This function has many keyword arguments that can be used to customize the final representation. For convenience some of the more frequently used kwargs have short aliases. See "Kwargs" for more details.

Parameters **data** (*object*) – an arbitrary python object to form the string "representation" of

Kwargs:

si, stritems, (bool): dict/list items use str instead of repr

strkeys, sk (bool): dict keys use str instead of repr

strvals, sv (bool): dict values use str instead of repr

nl, newlines (int | bool): number of top level nestings to place a newline after. If true all items are followed by newlines regardless of nesting level. Defaults to 1 for lists and True for dicts.

nobr, nobraces (bool, default=False): if True, text will not contain outer braces for containers

cbr, compact_brace (bool, default=False): if True, braces are compactified (i.e. they will not have newlines placed directly after them, think java / K&R / 1TBS)

trailsep, trailing_sep (bool): if True, a separator is placed after the last item in a sequence. By default this is True if there are any `nl > 0`.

explicit (bool, default=False): changes dict representation from `{k1: v1, ...}` to `dict(k1=v1, ...)`.

Modifies: default kvsep is modified to `'='` dict braces from *{}* to *dict()*.

compact (bool, default=False): Produces values more suitable for space constrianed environments

Modifies: default kvsep is modified to `'='` default itemsep is modified to `''` default nobraces is modified to `1`. default nl is modified to `0`. default strkeys to `True` default strvals to `True`

precision (int, default=None): if specified floats are formatted with this precision

kvsep (str, default=': '): separator between keys and values

itemsep (str, default=' '): separator between items

sort (bool | callable, default=None): if None, then sort unordered collections, but keep the ordering of ordered collections. This option attempts to be deterministic in most cases.

New in 0.8.0: if `sort` is callable, it will be used as a key-function to sort all collections.

if False, then nothing will be sorted, and the representation of unordered collections will be arbitrary and possibly non-determenistic.

if True, attempts to sort all collections in the returned text. Currently if True this WILL sort lists. Currently if True this WILL NOT sort OrderedDicts.

NOTE: The previous behavior may not be intuitive, as such the behavior of this arg is subject to change.

suppress_small (bool): passed to `numpy.array2string()` for ndarrays

max_line_width (int): passed to `numpy.array2string()` for ndarrays

with_dtype (bool): only relevant to numpy.ndarrays. if True includes the dtype. Defaults to *not strvals*.

align (bool | str, default=False): if True, will align multi-line dictionaries by the kvsep

extensions (FormatterExtensions): a custom *`FormatterExtensions`* instance that can overwrite or define how different types of objects are formatted.

Returns outstr - output string

Return type str

Note: There are also internal kwargs, which should not be used:

_return_info (bool): return information about child context

_root_info (depth): information about parent context

RelatedWork: `rich.pretty.pretty_repr()` `pprint.pformat()`

Example

```
>>> import ubelt as ub
>>> dict_ = {
...     'custom_types': [slice(0, 1, None), 1/3],
...     'nest_dict': {'k1': [1, 2, {3: {4, 5}}],
...                   'key2': [1, 2, {3: {4, 5}}],
...                   'key3': [1, 2, {3: {4, 5}}],
...                   },
...     'nest_dict2': {'k': [1, 2, {3: {4, 5}}]},
...     'nested_tuples': [tuple([1]), tuple([2, 3]), frozenset([4, 5, 6])],
...     'one_tup': tuple([1]),
...     'simple_dict': {'spam': 'eggs', 'ham': 'jam'},
...     'simple_list': [1, 2, 'red', 'blue'],
...     'odict': ub.odict([(1, '1'), (2, '2')]),
... }
>>> # In the interest of saving space we are only going to show the
>>> # output for the first example.
>>> result = ub.repr2(dict_, nl=1, precision=2)
>>> print(result)
{
    'custom_types': [slice(0, 1, None), 0.33],
    'nest_dict': {'k1': [1, 2, {3: {4, 5}}], 'key2': [1, 2, {3: {4, 5}}], 'key3':␣
↪[1, 2, {3: {4, 5}}]},
    'nest_dict2': {'k': [1, 2, {3: {4, 5}}]},
    'nested_tuples': [(1,), (2, 3), {4, 5, 6}],
```

(continues on next page)

(continued from previous page)

```
    'odict': {1: '1', 2: '2'},
    'one_tup': (1,),
    'simple_dict': {'ham': 'jam', 'spam': 'eggs'},
    'simple_list': [1, 2, 'red', 'blue'],
}
>>> # You can try the rest yourself.
>>> result = ub.repr2(dict_, nl=3, precision=2); print(result)
>>> result = ub.repr2(dict_, nl=2, precision=2); print(result)
>>> result = ub.repr2(dict_, nl=1, precision=2, itemsep='', explicit=True);␣
↪print(result)
>>> result = ub.repr2(dict_, nl=1, precision=2, nobr=1, itemsep='', explicit=True);␣
↪print(result)
>>> result = ub.repr2(dict_, nl=3, precision=2, cbr=True); print(result)
>>> result = ub.repr2(dict_, nl=3, precision=2, si=True); print(result)
>>> result = ub.repr2(dict_, nl=3, sort=True); print(result)
>>> result = ub.repr2(dict_, nl=3, sort=False, trailing_sep=False); print(result)
>>> result = ub.repr2(dict_, nl=3, sort=False, trailing_sep=False, nobr=True);␣
↪print(result)
```

Example

```
>>> import ubelt as ub
>>> def _nest(d, w):
...     if d == 0:
...         return {}
...     else:
...         return {'n{}'.format(d): _nest(d - 1, w + 1), 'm{}'.format(d): _nest(d -
↪ 1, w + 1)}
>>> dict_ = _nest(d=4, w=1)
>>> result = ub.repr2(dict_, nl=6, precision=2, cbr=1)
>>> print('---')
>>> print(result)
>>> result = ub.repr2(dict_, nl=-1, precision=2)
>>> print('---')
>>> print(result)
```

Example

```
>>> import ubelt as ub
>>> data = {'a': 100, 'b': [1, '2', 3], 'c': {20:30, 40: 'five'}}
>>> print(ub.repr2(data, nl=1))
{
    'a': 100,
    'b': [1, '2', 3],
    'c': {20: 30, 40: 'five'},
}
>>> # Compact is useful for things like timerit.Timerit labels
>>> print(ub.repr2(data, compact=True))
a=100,b=[1,2,3],c={20=30,40=five}
```

(continues on next page)

(continued from previous page)

```
>>> print(ub.repr2(data, compact=True, nobr=False))
{a=100,b=[1,2,3],c={20=30,40=five}}
```

ubelt.**shrinkuser**(*path*, *home='~'*)

Inverse of os.path.expanduser().

Parameters

- **path** (*str | PathLike*) – path in system file structure
- **home** (*str*) – symbol used to replace the home path. Defaults to '~', but you might want to use '$HOME' or '%USERPROFILE%' instead.

Returns path - shortened path replacing the home directory with a symbol

Return type str

Example

```
>>> from ubelt.util_path import *  # NOQA
>>> path = expanduser('~')
>>> assert path != '~'
>>> assert shrinkuser(path) == '~'
>>> assert shrinkuser(path + '1') == path + '1'
>>> assert shrinkuser(path + '/1') == join('~', '1')
>>> assert shrinkuser(path + '/1', '$HOME') == join('$HOME', '1')
>>> assert shrinkuser('.') == '.'
```

ubelt.**sorted_keys**(*dict_*, *key=None*, *reverse=False*)

Return an ordered dictionary sorted by its keys

Parameters

- **dict_** (*Dict[KT, VT]*) – dictionary to sort. The keys must be of comparable types.
- **key** (*Callable[[KT], Any] | None*) – If given as a callable, customizes the sorting by ordering using transformed keys.
- **reverse** (*bool, default=False*) – if True returns in descending order

Returns new dictionary where the keys are ordered

Return type OrderedDict[KT, VT]

Example

```
>>> import ubelt as ub
>>> dict_ = {'spam': 2.62, 'eggs': 1.20, 'jam': 2.92}
>>> newdict = sorted_keys(dict_)
>>> print(ub.repr2(newdict, nl=0))
{'eggs': 1.2, 'jam': 2.92, 'spam': 2.62}
>>> newdict = sorted_keys(dict_, reverse=True)
>>> print(ub.repr2(newdict, nl=0))
{'spam': 2.62, 'jam': 2.92, 'eggs': 1.2}
>>> newdict = sorted_keys(dict_, key=lambda x: sum(map(ord, x)))
```

(continues on next page)

(continued from previous page)

```
>>> print(ub.repr2(newdict, nl=0))
{'jam': 2.92, 'eggs': 1.2, 'spam': 2.62}
```

ubelt.**sorted_vals**(*dict_*, *key=None*, *reverse=False*)

Return an ordered dictionary sorted by its values

Parameters

- **dict_** (*Dict[KT, VT]*) – dictionary to sort. The values must be of comparable types.
- **key** (*Callable[[VT], Any] | None*) – If given as a callable, customizes the sorting by ordering using transformed values.
- **reverse** (*bool, default=False*) – if True returns in descending order

Returns new dictionary where the values are ordered

Return type OrderedDict[KT, VT]

Example

```
>>> import ubelt as ub
>>> dict_ = {'spam': 2.62, 'eggs': 1.20, 'jam': 2.92}
>>> newdict = sorted_vals(dict_)
>>> print(ub.repr2(newdict, nl=0))
{'eggs': 1.2, 'spam': 2.62, 'jam': 2.92}
>>> newdict = sorted_vals(dict_, reverse=True)
>>> print(ub.repr2(newdict, nl=0))
{'jam': 2.92, 'spam': 2.62, 'eggs': 1.2}
>>> newdict = sorted_vals(dict_, key=lambda x: x % 1.6)
>>> print(ub.repr2(newdict, nl=0))
{'spam': 2.62, 'eggs': 1.2, 'jam': 2.92}
```

ubelt.**split_archive**(*fpath*, *ext='.zip'*)

If fpath specifies a file inside a zipfile, it breaks it into two parts the path to the zipfile and the internal path in the zipfile.

Example

```
>>> split_archive('/a/b/foo.txt')
>>> split_archive('/a/b/foo.zip/bar.txt')
>>> split_archive('/a/b/foo.zip/baz/biz.zip/bar.py')
>>> split_archive('archive.zip')
>>> import ubelt as ub
>>> split_archive(ub.Path('/a/b/foo.zip/baz/biz.zip/bar.py'))
>>> split_archive('/a/b/foo.zip/baz.pt/bar.zip/bar.zip', '.pt')
```

Todo: Fix got/want for win32

(None, None) ('/a/b/foo.zip', 'bar.txt') ('/a/b/foo.zip/baz/biz.zip', 'bar.py') ('archive.zip', None) ('/a/b/foo.zip/baz/biz.zip', 'bar.py') ('/a/b/foo.zip/baz.pt', 'bar.zip/bar.zip')

`ubelt.split_modpath(modpath, check=True)`

Splits the modpath into the dir that must be in PYTHONPATH for the module to be imported and the modulepath relative to this directory.

Parameters

- **modpath** (*str*) – module filepath
- **check** (*bool*) – if False, does not raise an error if modpath is a directory and does not contain an `__init__.py` file.

Returns (directory, rel_modpath)

Return type Tuple[str, str]

Raises `ValueError` – if modpath does not exist or is not a package

Example

```
>>> from xdoctest import static_analysis
>>> modpath = static_analysis.__file__.replace('.pyc', '.py')
>>> modpath = abspath(modpath)
>>> dpath, rel_modpath = split_modpath(modpath)
>>> recon = join(dpath, rel_modpath)
>>> assert recon == modpath
>>> assert rel_modpath == join('xdoctest', 'static_analysis.py')
```

`ubelt.symlink(real_path, link_path, overwrite=False, verbose=0)`

Create a link `link_path` that mirrors `real_path`.

This function attempts to create a real symlink, but will fall back on a hard link or junction if symlinks are not supported.

Parameters

- **path** (*str | PathLike*) – path to real file or directory
- **link_path** (*str | PathLike*) – path to desired location for symlink
- **overwrite** (*bool, default=False*) – overwrite existing symlinks. This will not overwrite real files on systems with proper symlinks. However, on older versions of windows junctions are indistinguishable from real files, so we cannot make this guarantee.
- **verbose** (*int, default=0*) – verbosity level

Returns link path

Return type str | PathLike

Note: On systems that do not contain support for symlinks (e.g. some versions / configurations of Windows), this function will fall back on hard links or junctions [WikiNTFSLinks], [WikiHardLink]. The differences between the two are explained in [WikiSymLink].

If symlinks are not available, then `link_path` and `real_path` must exist on the same filesystem. Given that, this function always works in the sense that (1) `link_path` will mirror the data from `real_path`, (2) updates to one will effect the other, and (3) no extra space will be used.

More details can be found in `ubelt._win32_links`. On systems that support symlinks (e.g. Linux), none of the above applies.

References

Example

```
>>> import ubelt as ub
>>> dpath = ub.ensure_app_cache_dir('ubelt', 'test_symlink0')
>>> real_path = join(dpath, 'real_file.txt')
>>> link_path = join(dpath, 'link_file.txt')
>>> [ub.delete(p) for p in [real_path, link_path]]
>>> ub.writeto(real_path, 'foo')
>>> result = symlink(real_path, link_path)
>>> assert ub.readfrom(result) == 'foo'
>>> [ub.delete(p) for p in [real_path, link_path]]
```

Example

```
>>> import ubelt as ub
>>> from os.path import dirname
>>> dpath = ub.ensure_app_cache_dir('ubelt', 'test_symlink1')
>>> ub.delete(dpath)
>>> ub.ensuredir(dpath)
>>> _dirstats(dpath)
>>> real_dpath = ub.ensuredir((dpath, 'real_dpath'))
>>> link_dpath = ub.augpath(real_dpath, base='link_dpath')
>>> real_path = join(dpath, 'afile.txt')
>>> link_path = join(dpath, 'afile.txt')
>>> [ub.delete(p) for p in [real_path, link_path]]
>>> ub.writeto(real_path, 'foo')
>>> result = symlink(real_dpath, link_dpath)
>>> assert ub.readfrom(link_path) == 'foo', 'read should be same'
>>> ub.writeto(link_path, 'bar')
>>> _dirstats(dpath)
>>> assert ub.readfrom(link_path) == 'bar', 'very bad bar'
>>> assert ub.readfrom(real_path) == 'bar', 'changing link did not change real'
>>> ub.writeto(real_path, 'baz')
>>> _dirstats(dpath)
>>> assert ub.readfrom(real_path) == 'baz', 'very bad baz'
>>> assert ub.readfrom(link_path) == 'baz', 'changing real did not change link'
>>> ub.delete(link_dpath, verbose=1)
>>> _dirstats(dpath)
>>> assert not exists(link_dpath), 'link should not exist'
>>> assert exists(real_path), 'real path should exist'
>>> _dirstats(dpath)
>>> ub.delete(dpath, verbose=1)
>>> _dirstats(dpath)
>>> assert not exists(real_path)
```

`ubelt.take`(*items*, *indices*, *default=NoParam*)

Lookup a subset of an indexable object using a sequence of indices.

The `items` input is usually a list or dictionary. When `items` is a list, this should be a sequence of integers. When `items` is a dict, this is a list of keys to lookup in that dictionary.

For dictionaries, a default may be specified as a placeholder to use if a key from `indices` is not in `items`.

Parameters

- **items** (*Sequence[VT] | Mapping[KT, VT]*) – An indexable object to select items from.
- **indices** (*Iterable[int | KT]*) – A sequence of indexes into `items`.
- **default** (*Any, default=NoParam*) – if specified `items` must support the `get` method.

Yields *VT* – a selected item within the list

SeeAlso: *ubelt.dict_subset()*

Note: `ub.take(items, indices)` is equivalent to `(items[i] for i in indices)` when `default` is unspecified.

Notes

This is based on the `numpy.take()` function, but written in pure python.

Do not confuse this with `more_itertools.take()`, the behavior is very different.

Example

```
>>> import ubelt as ub
>>> items = [0, 1, 2, 3]
>>> indices = [2, 0]
>>> list(ub.take(items, indices))
[2, 0]
```

Example

```
>>> import ubelt as ub
>>> dict_ = {1: 'a', 2: 'b', 3: 'c'}
>>> keys = [1, 2, 3, 4, 5]
>>> result = list(ub.take(dict_, keys, None))
>>> assert result == ['a', 'b', 'c', None, None]
```

Example

```
>>> import ubelt as ub
>>> dict_ = {1: 'a', 2: 'b', 3: 'c'}
>>> keys = [1, 2, 3, 4, 5]
>>> try:
>>>     print(list(ub.take(dict_, keys)))
>>>     raise AssertionError('did not get key error')
>>> except KeyError:
>>>     print('correctly got key error')
```

`ubelt.timeparse(stamp, allow_dateutil=True)`

Create a `datetime.datetime` object from a string timestamp.

Without any extra dependencies this will parse the output of `ubelt.util_time.timestamp()` into a datetime object. In the case where the format differs, *dateutil.parser.parse* will be used if the *python-dateutil* package is installed.

Parameters

- **stamp** (*str*) – a string encoded timestamp
- **allow_dateutil** (*bool*) – if False we only use the minimal parsing and do not allow a fallback to dateutil.

Returns the parsed datetime

Return type datetime.datetime

Raises `ValueError` – if if parsing fails.

Todo:

- [ ] Allow defaulting to local or utm timezone (currently default is local)

Example

```
>>> import ubelt as ub
>>> # Demonstrate a round trip of timestamp and timeparse
>>> stamp = ub.timestamp()
>>> datetime = ub.timeparse(stamp)
>>> assert ub.timestamp(datetime) == stamp
>>> # Round trip with precision
>>> stamp = ub.timestamp(precision=4)
>>> datetime = ub.timeparse(stamp)
>>> assert ub.timestamp(datetime, precision=4) == stamp
```

Example

```
>>> import ubelt as ub
>>> # We should always be able to parse these
>>> good_stamps = [
>>>     '2000-11-22T111111.44444Z',
>>>     '2000-11-22T111111.44444+5',
>>>     '2000-11-22T111111.44444-05',
>>>     '2000-11-22T111111.44444-0500',
>>>     '2000-11-22T111111.44444+0530',
>>>     '2000-11-22T111111Z',
>>>     '2000-11-22T111111+5',
>>>     '2000-11-22T111111+0530',
>>> ]
>>> for stamp in good_stamps:
>>>     print(f'----')
>>>     print(f'stamp={stamp}')
```

(continues on next page)

(continued from previous page)

```
>>>     result = ub.timeparse(stamp, allow_dateutil=0)
>>>     print(f'result={result!r}')
>>>     recon = ub.timestamp(result)
>>>     print(f'recon={recon}')
```

Example

```
>>> import ubelt as ub
>>> # We require dateutil to handle these types of stamps
>>> import pytest
>>> conditional_stamps = [
>>>         '2000-01-02T11:23:58.12345+5:30',
>>>         '09/25/2003',
>>>         'Thu Sep 25 10:36:28 2003',
>>> ]
>>> for stamp in conditional_stamps:
>>>     with pytest.raises(ValueError):
>>>         result = ub.timeparse(stamp, allow_dateutil=False)
>>> have_dateutil = bool(ub.modname_to_modpath('dateutil'))
>>> if have_dateutil:
>>>     for stamp in conditional_stamps:
>>>         result = ub.timeparse(stamp)
```

`ubelt.timestamp`(*datetime=None*, *precision=0*, *method='iso8601'*)

Make a concise iso8601 timestamp suitable for use in filenames.

Parameters

- **datetime** (*datetime.datetime | None*) – A datetime to format into a timestamp. If unspecified, the current local time is used.
- **method** (*str*) – Type of timestamp. Currently the only option is iso8601. This argument may be removed in the future.
- **precision** (*int*) – if non-zero, adds up to 6 digits of sub-second precision.

Returns stamp

Return type str

Note: For more info see [WikiISO8601], [PyStrptime], [PyTime].

References

Todo:

- [ ] Allow defaulting to local or utm timezone (currently default is local)

Example

```
>>> import ubelt as ub
>>> stamp = ub.timestamp()
>>> print('stamp = {!r}'.format(stamp))
stamp = ...-...-...T...
```

Example

```
>>> import ubelt as ub
>>> import datetime as datetime_mod
>>> from datetime import datetime as datetime_cls
>>> # Create a datetime object with timezone information
>>> ast_tzinfo = datetime_mod.timezone(datetime_mod.timedelta(hours=-4), 'AST')
>>> datetime = datetime_cls.utcfromtimestamp(123456789.123456789).
↪replace(tzinfo=ast_tzinfo)
>>> stamp = ub.timestamp(datetime, precision=2)
>>> print('stamp = {!r}'.format(stamp))
stamp = '1973-11-29T213309.12-4'
```

```
>>> # Demo with a fractional hour timezone
>>> act_tzinfo = datetime_mod.timezone(datetime_mod.timedelta(hours=+9.5), 'ACT')
>>> datetime = datetime_cls.utcfromtimestamp(123456789.123456789).
↪replace(tzinfo=act_tzinfo)
>>> stamp = ub.timestamp(datetime, precision=2)
>>> print('stamp = {!r}'.format(stamp))
stamp = '1973-11-29T213309.12+0930'
```

Example

```
>>> # xdoctest: +REQUIRES(module:dateutil)
>>> # Make sure we are compatible with dateutil
>>> import ubelt as ub
>>> from dateutil.tz import tzlocal
>>> import datetime as datetime_mod
>>> from datetime import datetime as datetime_cls
>>> tzinfo_list = [
>>>     datetime_mod.timezone(datetime_mod.timedelta(hours=+9.5), 'ACT'),
>>>     datetime_mod.timezone(datetime_mod.timedelta(hours=-4), 'AST'),
>>>     datetime_mod.timezone(datetime_mod.timedelta(hours=0), 'UTC'),
>>>     datetime_mod.timezone.utc,
>>>     None,
>>>     tzlocal()
>>> ]
>>> # Note: there is a win32 bug here
>>> # https://bugs.python.org/issue37 that means we cant use
>>> # dates close to the epoch
>>> datetime_list = [
>>>     datetime_cls.utcfromtimestamp(123456789.123456789 + 315360000),
>>>     datetime_cls.utcfromtimestamp(0 + 315360000),
```

(continues on next page)

(continued from previous page)

```
>>> ]
>>> basis = {
>>>     'precision': [0, 3, 9],
>>>     'tzinfo': tzinfo_list,
>>>     'datetime': datetime_list,
>>> }
>>> for params in ub.named_product(basis):
>>>     dtime = params['datetime'].replace(tzinfo=params['tzinfo'])
>>>     precision = params.get('precision', 0)
>>>     stamp = ub.timestamp(datetime=dtime, precision=precision)
>>>     recon = ub.timeparse(stamp)
>>>     alt = recon.strftime('%Y-%m-%dT%H%M%S.%f%z')
>>>     print('---')
>>>     print('params = {}'.format(ub.repr2(params, nl=1)))
>>>     print(f'dtime={dtime}')
>>>     print(f'stamp={stamp}')
>>>     print(f'recon={recon}')
>>>     print(f'alt  ={alt}')
>>>     shift = 10 ** precision
>>>     a = int(dtime.timestamp() * shift)
>>>     b = int(recon.timestamp() * shift)
>>>     assert a == b, f'{a} != {b}'
```

`ubelt.touch`(*fpath*, *mode=438*, *dir_fd=None*, *verbose=0*, ***kwargs*)

change file timestamps

Works like the touch unix utility

Parameters

- **fpath** (*str* | *PathLike*) – name of the file
- **mode** (*int*) – file permissions (python3 and unix only)
- **dir_fd** (*io.IOBase*) – optional directory file descriptor. If specified, fpath is interpreted as relative to this descriptor (python 3 only).
- **verbose** (*int*) – verbosity
- ****kwargs** – extra args passed to `os.utime()` (python 3 only).

Returns path to the file

Return type str

References

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> dpath = ub.ensure_app_cache_dir('ubelt')
>>> fpath = join(dpath, 'touch_file')
>>> assert not exists(fpath)
>>> ub.touch(fpath)
```

(continues on next page)

(continued from previous page)

```
>>> assert exists(fpath)
>>> os.unlink(fpath)
```

`ubelt.unique(items, key=None)`

Generates unique items in the order they appear.

Parameters

- **items** (*Iterable[T]*) – list of items
- **key** (*Callable[[T], Any], default=None*) – custom normalization function. If specified returns items where `key(item)` is unique.

Yields *T* – a unique item from the input sequence

Notes

Functionally equivalent to `more_itertools.unique_everseen()`.

Example

```
>>> import ubelt as ub
>>> items = [4, 6, 6, 0, 6, 1, 0, 2, 2, 1]
>>> unique_items = list(ub.unique(items))
>>> assert unique_items == [4, 6, 0, 1, 2]
```

Example

```
>>> import ubelt as ub
>>> items = ['A', 'a', 'b', 'B', 'C', 'c', 'D', 'e', 'D', 'E']
>>> unique_items = list(ub.unique(items, key=str.lower))
>>> assert unique_items == ['A', 'b', 'C', 'D', 'e']
>>> unique_items = list(ub.unique(items))
>>> assert unique_items == ['A', 'a', 'b', 'B', 'C', 'c', 'D', 'e', 'E']
```

`ubelt.unique_flags(items, key=None)`

Returns a list of booleans corresponding to the first instance of each unique item.

Parameters

- **items** (*Sequence[VT]*) – indexable collection of items
- **key** (*Callable[[VT], Any] | None, default=None*) – custom normalization function. If specified returns items where `key(item)` is unique.

Returns flags the items that are unique

Return type List[bool]

Example

```
>>> import ubelt as ub
>>> items = [0, 2, 1, 1, 0, 9, 2]
>>> flags = ub.unique_flags(items)
>>> assert flags == [True, True, True, False, False, True, False]
>>> flags = ub.unique_flags(items, key=lambda x: x % 2 == 0)
>>> assert flags == [True, False, True, False, False, False, False]
```

`ubelt.userhome(username=None)`

Returns the path to some user's home directory.

Parameters **username** (*str | None*) – name of a user on the system. If not specified, the current user is inferred.

Returns userhome_dpath - path to the specified home directory

Return type str

Raises

- `KeyError` – if the specified user does not exist on the system
- `OSError` – if username is unspecified and the current user cannot be inferred

Example

```
>>> from ubelt.util_path import *  # NOQA
>>> import getpass
>>> username = getpass.getuser()
>>> assert userhome() == expanduser('~')
>>> assert userhome(username) == expanduser('~')
```

`ubelt.varied_values(longform, min_variations=0, default=NoParam)`

Given a list of dictionaries, find the values that differ between them.

Parameters

- **longform** (*List[Dict[KT, VT]]*) – This is longform data, as described in [SeabornLongform]. It is a list of dictionaries.

  Each item in the list - or row - is a dictionary and can be thought of as an observation. The keys in each dictionary are the columns. The values of the dictionary must be hashable. Lists will be converted into tuples.

- **min_variations** (*int, default=0*) – "columns" with fewer than `min_variations` unique values are removed from the result.
- **default** (*VT, default=NoParam*) – if specified, unspecified columns are given this value.

Returns a mapping from each "column" to the set of unique values it took over each "row". If a column is not specified for each row, it is assumed to take a *default* value, if it is specified.

Return type Dict[KT, List[VT]]

Raises `KeyError` – If `default` is unspecified and all the rows do not contain the same columns.

References

Example

```
>>> # An example use case is to determine what values of a
>>> # configuration dictionary were tried in a random search
>>> # over a parameter grid.
>>> import ubelt as ub
>>> longform = [
>>>     {'col1': 1, 'col2': 'foo', 'col3': None},
>>>     {'col1': 1, 'col2': 'foo', 'col3': None},
>>>     {'col1': 2, 'col2': 'bar', 'col3': None},
>>>     {'col1': 3, 'col2': 'bar', 'col3': None},
>>>     {'col1': 9, 'col2': 'bar', 'col3': None},
>>>     {'col1': 1, 'col2': 'bar', 'col3': None},
>>> ]
>>> varied = ub.varied_values(longform)
>>> print('varied = {}'.format(ub.repr2(varied, nl=1)))
varied = {
    'col1': {1, 2, 3, 9},
    'col2': {'bar', 'foo'},
    'col3': {None},
}
```

Example

```
>>> import ubelt as ub
>>> import random
>>> longform = [
>>>     {'col1': 1, 'col2': 'foo', 'col3': None},
>>>     {'col1': 1, 'col2': [1, 2], 'col3': None},
>>>     {'col1': 2, 'col2': 'bar', 'col3': None},
>>>     {'col1': 3, 'col2': 'bar', 'col3': None},
>>>     {'col1': 9, 'col2': 'bar', 'col3': None},
>>>     {'col1': 1, 'col2': 'bar', 'col3': None, 'extra_col': 3},
>>> ]
>>> # Operation fails without a default
>>> import pytest
>>> with pytest.raises(KeyError):
>>>     varied = ub.varied_values(longform)
>>> #
>>> # Operation works with a default
>>> varied = ub.varied_values(longform, default='<unset>')
>>> expected = {
>>>     'col1': {1, 2, 3, 9},
>>>     'col2': {'bar', 'foo', (1, 2)},
>>>     'col3': set([None]),
>>>     'extra_col': {'<unset>', 3},
>>> }
>>> print('varied = {!r}'.format(varied))
>>> assert varied == expected
```

Example

```
>>> # xdoctest: +REQUIRES(PY3)
>>> # Random numbers are different in Python2, so skip in that case
>>> import ubelt as ub
>>> import random
>>> num_cols = 11
>>> num_rows = 17
>>> rng = random.Random(0)
>>> # Generate a set of columns
>>> columns = sorted(ub.hash_data(i)[0:8] for i in range(num_cols))
>>> # Generate rows for each column
>>> longform = [
>>>     {key: ub.hash_data(key)[0:8] for key in columns}
>>>     for _ in range(num_rows)
>>> ]
>>> # Add in some varied values in random positions
>>> for row in longform:
>>>     if rng.random() > 0.5:
>>>         for key in sorted(row.keys()):
>>>             if rng.random() > 0.95:
>>>                 row[key] = 'special-' + str(rng.randint(1, 32))
>>> varied = ub.varied_values(longform, min_variations=1)
>>> print('varied = {}'.format(ub.repr2(varied, nl=1, sort=True)))
varied = {
    '095f3e44': {'8fb4d4c9', 'special-23'},
    '365d11a1': {'daa409da', 'special-31', 'special-32'},
    '5815087d': {'1b823610', 'special-3'},
    '7b54b668': {'349a782c', 'special-10'},
    'b8244d02': {'d57bca90', 'special-8'},
    'f27b5bf8': {'fa0f90d1', 'special-19'},
}
```

`ubelt.writeto`(*fpath*, *to_write*, *aslines=False*, *verbose=None*)

Writes (utf8) text to a file.

Parameters

- **fpath** (*str | PathLike*) – file path
- **to_write** (*str*) – text to write (must be unicode text)
- **aslines** (*bool*) – if True to_write is assumed to be a list of lines
- **verbose** (*bool*) – verbosity flag

Note: In CPython you may want to use `open(<fpath>).write(<to_write>)` instead. This function exists as a convenience for writing in Python2. After 2020-01-01, we may consider deprecating the function.

NOTE: In PyPy `open(<fpath>).write(<to_write>)` does not work. See *https://pypy.org/compat.html*. This is an argument for keeping this function.

NOTE: With modern versions of Python, it is generally recommened to use `pathlib.Path.write_text()` instead. Although there does seem to be some corner case this handles better on win32, so maybe useful?

Example

```
>>> import ubelt as ub
>>> import os
>>> from os.path import exists
>>> dpath = ub.ensure_app_cache_dir('ubelt')
>>> fpath = dpath + '/' + 'testwrite.txt'
>>> if exists(fpath):
>>>     os.remove(fpath)
>>> to_write = 'utf-8 symbols , , , , , , , , and .'
>>> ub.writeto(fpath, to_write)
>>> read_ = ub.readfrom(fpath)
>>> print('read_    = ' + read_)
>>> print('to_write = ' + to_write)
>>> assert read_ == to_write
```

Example

```
>>> import ubelt as ub
>>> import os
>>> from os.path import exists
>>> dpath = ub.ensure_app_cache_dir('ubelt')
>>> fpath = dpath + '/' + 'testwrite2.txt'
>>> if exists(fpath):
>>>     os.remove(fpath)
>>> to_write = ['a\n', 'b\n', 'c\n', 'd\n']
>>> ub.writeto(fpath, to_write, aslines=True)
>>> read_ = ub.readfrom(fpath, aslines=True)
>>> print('read_    = {}'.format(read_))
>>> print('to_write = {}'.format(to_write))
>>> assert read_ == to_write
```

Example

```
>>> # With modern Python, use pathlib.Path (or ub.Path) instead
>>> import ubelt as ub
>>> dpath = ub.Path.appdir('ubelt/tests/io').ensuredir()
>>> fpath = (dpath / 'test_file.txt').delete()
>>> to_write = 'utf-8 symbols , , , , , , , , and .'
>>> ub.writeto(fpath, to_write)
>>> fpath.write_bytes(to_write.encode('utf8'))
>>> assert fpath.read_bytes().decode('utf8') == to_write
```

class ubelt.**zopen**(*fpath*, *mode='r'*, *seekable=False*, *ext='.zip'*)

Bases: *ubelt.util_mixins.NiceRepr*

An abstraction of the normal `open()` function that can also handle reading data directly inside of zipfiles.

This is a file-object like interface [FileObj] — i.e. it supports the read and write methods to an underlying resource.

Can open a file normally or open a file within a zip file (readonly). Tries to read from memory only, but will extract to a tempfile if necessary.

Just treat the zipfile like a directory, e.g. /path/to/myzip.zip/compressed/path.txt OR? e.g. /path/to/myzip.zip:compressed/path.txt

References

Todo:

- **[ ] Fast way to open a base zipfile, query what is inside, and** then choose a file to further zopen (and passing along the same open zipfile reference maybe?).
- [ ] Write mode in some restricted setting?

Parameters

- **fpath** (*str | PathLike*) – path to a file, or a special path that denotes both a path to a zipfile and a path to a archived file inside of the zipfile.
- **mode** (*str*) – Currently only "r" - readonly mode is supported
- **seekable** (*bool*) – If True, attempts to force "seekability" of the underlying file-object, for compressed files this will first extract the file to a temporary location on disk. If False, any underlying compressed file will be opened directly which may result in the object being non-seekable.
- **ext** (*str*) – The extension of the zipfile. Modify this is a non-standard extension is used (e.g. for torch packages).

Example

```
>>> from ubelt.util_zip import *  # NOQA
>>> import pickle
>>> import ubelt as ub
>>> dpath = ub.ensure_app_cache_dir('ubelt/tests/util_zip')
>>> dpath = ub.Path(dpath)
>>> data_fpath = dpath / 'test.pkl'
>>> data = {'demo': 'data'}
>>> with open(str(data_fpath), 'wb') as file:
>>>     pickle.dump(data, file)
>>> # Write data
>>> import zipfile
>>> zip_fpath = dpath / 'test_zip.archive'
>>> stl_w_zfile = zipfile.ZipFile(os.fspath(zip_fpath), mode='w')
>>> stl_w_zfile.write(os.fspath(data_fpath), os.fspath(data_fpath.relative_
↪to(dpath)))
>>> stl_w_zfile.close()
>>> stl_r_zfile = zipfile.ZipFile(os.fspath(zip_fpath), mode='r')
>>> stl_r_zfile.namelist()
>>> stl_r_zfile.close()
>>> # Test zopen
>>> self = zopen(zip_fpath / 'test.pkl', mode='rb', ext='.archive')
```

(continues on next page)

(continued from previous page)

```
>>> print(self._split_archive())
>>> print(self.namelist())
>>> self.close()
>>> self = zopen(zip_fpath / 'test.pkl', mode='rb', ext='.archive')
>>> recon1 = pickle.loads(self.read())
>>> self.close()
>>> self = zopen(zip_fpath / 'test.pkl', mode='rb', ext='.archive')
>>> recon2 = pickle.load(self)
>>> self.close()
>>> assert recon1 == recon2
>>> assert recon1 is not recon2
```

Example

```
>>> # Test we can load json data from a zipfile
>>> from ubelt.util_zip import *  # NOQA
>>> import ubelt as ub
>>> import json
>>> import zipfile
>>> dpath = ub.ensure_app_cache_dir('ubelt/tests/util_zip')
>>> infopath = join(dpath, 'info.json')
>>> ub.writeto(infopath, '{"x": "1"}')
>>> zippath = join(dpath, 'infozip.zip')
>>> internal = 'folder/info.json'
>>> with zipfile.ZipFile(zippath, 'w') as myzip:
>>>     myzip.write(infopath, internal)
>>> fpath = zippath + '/' + internal
>>> # Test context manager
>>> with zopen(fpath, 'r') as self:
>>>     info2 = json.load(self)
>>>     assert info2['x'] == '1'
>>> # Test outside of context manager
>>> self = zopen(fpath, 'r')
>>> print(self._split_archive())
>>> info2 = json.load(self)
>>> assert info2['x'] == '1'
>>> # Test nice repr (with zfile)
>>> print('self = {!r}'.format(self))
>>> self.close()
```

Example

```
>>> # Coverage tests --- move to unit-test
>>> from ubelt.util_zip import *  # NOQA
>>> import ubelt as ub
>>> import json
>>> import zipfile
>>> dpath = ub.ensure_app_cache_dir('ubelt/tests/util_zip')
>>> textpath = join(dpath, 'seekable_test.txt')
```

(continues on next page)

(continued from previous page)

```
>>> text = chr(10).join(['line{}'.format(i) for i in range(10)])
>>> ub.writeto(textpath, text)
>>> zippath = join(dpath, 'seekable_test.zip')
>>> internal = 'folder/seekable_test.txt'
>>> with zipfile.ZipFile(zippath, 'w') as myzip:
>>>     myzip.write(textpath, internal)
>>> ub.delete(textpath)
>>> fpath = zippath + '/' + internal
>>> # Test seekable
>>> self_seekable = zopen(fpath, 'r', seekable=True)
>>> assert self_seekable.seekable()
>>> self_seekable.seek(8)
>>> assert self_seekable.readline() == 'ne1' + chr(10)
>>> assert self_seekable.readline() == 'line2' + chr(10)
>>> self_seekable.seek(8)
>>> assert self_seekable.readline() == 'ne1' + chr(10)
>>> assert self_seekable.readline() == 'line2' + chr(10)
>>> # Test non-seekable?
>>> # Sometimes non-seekable files are still seekable
>>> maybe_seekable = zopen(fpath, 'r', seekable=False)
>>> if maybe_seekable.seekable():
>>>     maybe_seekable.seek(8)
>>>     assert maybe_seekable.readline() == 'ne1' + chr(10)
>>>     assert maybe_seekable.readline() == 'line2' + chr(10)
>>>     maybe_seekable.seek(8)
>>>     assert maybe_seekable.readline() == 'ne1' + chr(10)
>>>     assert maybe_seekable.readline() == 'line2' + chr(10)
```

Example

```
>>> # More coverage tests --- move to unit-test
>>> from ubelt.util_zip import *  # NOQA
>>> import ubelt as ub
>>> import pytest
>>> dpath = ub.ensure_app_cache_dir('ubelt/tests/util_zip')
>>> with pytest.raises(OSError):
>>>     self = zopen('', 'r')
>>> # Test open non-zip exsting file
>>> existing_fpath = join(dpath, 'exists.json')
>>> ub.writeto(existing_fpath, '{"x": "1"}')
>>> self = zopen(existing_fpath, 'r')
>>> assert self.read() == '{"x": "1"}'
>>> # Test dir
>>> dir(self)
>>> # Test nice
>>> print(self)
>>> print('self = {!r}'.format(self))
>>> self.close()
>>> # Test open non-zip non-existing file
>>> nonexisting_fpath = join(dpath, 'does-not-exist.txt')
```

(continues on next page)

(continued from previous page)

```
>>> ub.delete(nonexisting_fpath)
>>> with pytest.raises(OSError):
>>>     self = zopen(nonexisting_fpath, 'r')
>>> with pytest.raises(NotImplementedError):
>>>     self = zopen(nonexisting_fpath, 'w')
>>> # Test nice-repr
>>> self = zopen(existing_fpath, 'r')
>>> print('self = {!r}'.format(self))
>>> # pathological
>>> self = zopen(existing_fpath, 'r')
>>> self._handle = None
>>> dir(self)
```

property zfile

Access the underlying archive file

namelist()

Lists the contents of this zipfile

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex

BIBLIOGRAPHY

[SO_11495783] https://stackoverflow.com/questions/11495783/redirect-subprocess-stderr-to-stdout

[SO_7729336] https://stackoverflow.com/questions/7729336/how-can-i-print-and-display-subprocess-stdout-and-stderr-output-withou

[SO_33560364] https://stackoverflow.com/questions/33560364/python-windows-parsing-command-lines-with-shlex

[NoColor] https://no-color.org/

[SO_651794] http://stackoverflow.com/questions/651794/init-dict-of-dicts

[minstrel271_namedproduct] https://gist.github.com/minstrel271/d51654af3fa4e6411267

[pytb_namedproduct] https://py-toolbox.readthedocs.io/en/latest/modules/itertools.html#

[Hettinger_namedproduct] https://twitter.com/raymondh/status/970380630822305792

[SeabornLongform] https://seaborn.pydata.org/tutorial/data_structure.html#long-form-data

[Shichao_2012] https://blog.shichao.io/2012/10/04/progress_speed_indicator_for_urlretrieve_in_python.html

[SO_15644964] http://stackoverflow.com/questions/15644964/python-progress-bar-and-downloads

[SO_16694907] http://stackoverflow.com/questions/16694907/how-to-download-large-file-in-python-with-requests-py

[TorchDL] https://github.com/pytorch/pytorch/blob/2787f1d8edbd4aadd4a8680d204341a1d7112e2d/torch/hub.py#L347

[SO_3431825] http://stackoverflow.com/questions/3431825/md5-checksum-of-a-file

[SO_5001893] http://stackoverflow.com/questions/5001893/when-to-use-sha-1-vs-sha-2

[SO_67631] https://stackoverflow.com/questions/67631/import-module-given-path

[SO_1158076] https://stackoverflow.com/questions/1158076/implement-touch-using-python

[WikiSymLink] https://en.wikipedia.org/wiki/Symbolic_link

[WikiHardLink] https://en.wikipedia.org/wiki/Hard_link

[WikiNTFSLinks] https://en.wikipedia.org/wiki/NTFS_links

[SO_434287] http://stackoverflow.com/questions/434287/iterate-over-a-list-in-chunks

[WikiMemoize] https://wiki.python.org/moin/PythonDecoratorLibrary#Memoize

[FunctoolsCache] https://docs.python.org/3/library/functools.html

[ActiveState_Miller_2010] http://code.activestate.com/recipes/577452-a-memoize-decorator-for-instance-methods

[estebistec_memoize] https://github.com/estebistec/python-memoized-property

[CPythonIssue21301] https://bugs.python.org/issue21301

[XDG_Spec] https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html

[SO_43853548] https://stackoverflow.com/questions/43853548/xdg-windows

[SO_11113974] https://stackoverflow.com/questions/11113974/cross-plat-path

[harawata_appdirs] https://github.com/harawata/appdirs#supported-directories

[AS_appdirs] https://github.com/ActiveState/appdirs

[SO_377017] https://stackoverflow.com/questions/377017/test-if-executable-exists-in-python/377028#377028

[shutil_which] https://docs.python.org/dev/library/shutil.html#shutil.which

[WikiISO8601] https://en.wikipedia.org/wiki/ISO_8601

[PyStrptime] https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior

[PyTime] https://docs.python.org/3/library/time.html

[FileObj] https://docs.python.org/3/glossary.html#term-file-object

[SO_651794] http://stackoverflow.com/questions/651794/init-dict-of-dicts

[CPythonIssue21301] https://bugs.python.org/issue21301

[SO_434287] http://stackoverflow.com/questions/434287/iterate-over-a-list-in-chunks

[SO_11495783] https://stackoverflow.com/questions/11495783/redirect-subprocess-stderr-to-stdout

[SO_7729336] https://stackoverflow.com/questions/7729336/how-can-i-print-and-display-subprocess-stdout-and-stderr-output-withou

[SO_33560364] https://stackoverflow.com/questions/33560364/python-windows-parsing-command-lines-with-shlex

[Shichao_2012] https://blog.shichao.io/2012/10/04/progress_speed_indicator_for_urlretrieve_in_python.html

[SO_15644964] http://stackoverflow.com/questions/15644964/python-progress-bar-and-downloads

[SO_16694907] http://stackoverflow.com/questions/16694907/how-to-download-large-file-in-python-with-requests-py

[TorchDL] https://github.com/pytorch/pytorch/blob/2787f1d8edbd4aadd4a8680d204341a1d7112e2d/torch/hub.py#L347

[SO_377017] https://stackoverflow.com/questions/377017/test-if-executable-exists-in-python/377028#377028

[shutil_which] https://docs.python.org/dev/library/shutil.html#shutil.which

[SO_3431825] http://stackoverflow.com/questions/3431825/md5-checksum-of-a-file

[SO_5001893] http://stackoverflow.com/questions/5001893/when-to-use-sha-1-vs-sha-2

[SO_67631] https://stackoverflow.com/questions/67631/import-module-given-path

[WikiMemoize] https://wiki.python.org/moin/PythonDecoratorLibrary#Memoize

[FunctoolsCache] https://docs.python.org/3/library/functools.html

[ActiveState_Miller_2010] http://code.activestate.com/recipes/577452-a-memoize-decorator-for-instance-methods

[estebistec_memoize] https://github.com/estebistec/python-memoized-property

[minstrel271_namedproduct] https://gist.github.com/minstrel271/d51654af3fa4e6411267

[pytb_namedproduct] https://py-toolbox.readthedocs.io/en/latest/modules/itertools.html#

[Hettinger_namedproduct] https://twitter.com/raymondh/status/970380630822305792

[WikiSymLink] https://en.wikipedia.org/wiki/Symbolic_link

[WikiHardLink] https://en.wikipedia.org/wiki/Hard_link

[WikiNTFSLinks] https://en.wikipedia.org/wiki/NTFS_links

[WikiISO8601] https://en.wikipedia.org/wiki/ISO_8601

[PyStrptime] https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior

[PyTime] https://docs.python.org/3/library/time.html

[SO_1158076] https://stackoverflow.com/questions/1158076/implement-touch-using-python

[SeabornLongform] https://seaborn.pydata.org/tutorial/data_structure.html#long-form-data

[FileObj] https://docs.python.org/3/glossary.html#term-file-object

PYTHON MODULE INDEX

u

INDEX

A

B

C

D

E

F

G

N

O

P

R

S

T

U

V

W

Z