UBelt Documentation

Release 0.9.5

Jon Crall

Feb 05, 2021

PACKAGE LAYOUT

1 The API by usefulness **3**

1.1 ubelt.orderedset . . . 5
1.2 ubelt.progiter . . . 5
1.3 ubelt.timerit . . . 5
1.4 ubelt.util_arg . . . 5
1.5 ubelt.util_cache . . . 5
1.6 ubelt.util_cmd . . . 5
1.7 ubelt.util_colors . . . 5
1.8 ubelt.util_const . . . 6
1.9 ubelt.util_dict . . . 6
1.10 ubelt.util_download . . . 6
1.11 ubelt.util_format . . . 6
1.12 ubelt.util_func . . . 6
1.13 ubelt.util_hash . . . 6
1.14 ubelt.util_import . . . 6
1.15 ubelt.util_io . . . 7
1.16 ubelt.util_links . . . 7
1.17 ubelt.util_list . . . 7
1.18 ubelt.util_memoize . . . 7
1.19 ubelt.util_mixins . . . 7
1.20 ubelt.util_path . . . 7
1.21 ubelt.util_platform . . . 7
1.22 ubelt.util_str . . . 8
1.23 ubelt.util_stream . . . 8
1.24 ubelt.util_time . . . 8
1.24.1 ubelt . . . 8
1.24.1.1 ubelt package . . . 8
1.24.1.1.1 Submodules . . . 8
1.24.1.1.1.1 ubelt.orderedset module . . . 8
1.24.1.1.1.2 ubelt.progiter module . . . 13
1.24.1.1.1.3 ubelt.timerit module . . . 19
1.24.1.1.1.4 ubelt.util_arg module . . . 24
1.24.1.1.1.5 ubelt.util_cache module . . . 26
1.24.1.1.1.6 ubelt.util_cmd module . . . 33
1.24.1.1.1.7 ubelt.util_colors module . . . 35
1.24.1.1.1.8 ubelt.util_const module . . . 37
1.24.1.1.1.9 ubelt.util_dict module . . . 37
1.24.1.1.1.10 ubelt.util_download module . . . 46
1.24.1.1.1.11 ubelt.util_format module . . . 50
1.24.1.1.1.12 ubelt.util_func module . . . 54

1.24.1.1.1.13 ubelt.util_hash module 55
1.24.1.1.1.14 ubelt.util_import module 59
1.24.1.1.1.15 ubelt.util_io module 64
1.24.1.1.1.16 ubelt.util_links module 66
1.24.1.1.1.17 ubelt.util_list module 68
1.24.1.1.1.18 ubelt.util_memoize module 77
1.24.1.1.1.19 ubelt.util_mixins module 81
1.24.1.1.1.20 ubelt.util_path module 83
1.24.1.1.1.21 ubelt.util_platform module 87
1.24.1.1.1.22 ubelt.util_str module 91
1.24.1.1.1.23 ubelt.util_stream module 94
1.24.1.1.1.24 ubelt.util_time module 96
1.24.1.1.2 Module contents 97

2 Indices and tables **177**

Python Module Index **179**

Index **181**

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.

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()	2140
ubelt.ProgIter()	715
ubelt.expandpath()	695
ubelt.ensuredir()	553
ubelt.take()	430
ubelt.odict()	391
ubelt.map_vals()	331
ubelt.dzip()	278
ubelt.NiceRepr()	261
ubelt.ddict()	255
ubelt.augpath()	229
ubelt.argflag()	209
ubelt.flatten()	202
ubelt.argval()	200
ubelt.cmd()	199
ubelt.peek()	196
ubelt.NoParam()	189
ubelt.Timerit()	187
ubelt.dict_hist()	173
ubelt.codeblock()	172
ubelt.group_items()	168
ubelt.iterable()	168
ubelt.hash_data()	142
ubelt.grabdata()	127
ubelt.color_text()	97
ubelt.delete()	94
ubelt.dict_subset()	89
ubelt.oset()	88
ubelt.compress()	87
ubelt.allsame()	86
ubelt.Cacher()	81
ubelt.Timer()	70
ubelt.dict_isect()	68
ubelt.indent()	60

continues on next page

Table 1 – continued from previous page

Function name	Usefulness
ubelt.argsort()	59
ubelt.chunks()	48
ubelt.map_keys()	48
ubelt.invert_dict()	47
ubelt.dict_union()	47
ubelt.memoize()	46
ubelt.timestamp()	46
ubelt.find_duplicates()	45
ubelt.unique()	43
ubelt.import_module_from_path()	40
ubelt.sorted_vals()	39
ubelt.dict_diff()	38
ubelt.hzcat()	38
ubelt.argmax()	37
ubelt.memoize_property()	37
ubelt.writeto()	37
ubelt.iter_window()	35
ubelt.readfrom()	34
ubelt.paragraph()	33
ubelt.identity()	33
ubelt.symlink()	32
ubelt.memoize_method()	31
ubelt.ensure_unicode()	24
ubelt.touch()	24
ubelt.hash_file()	24
ubelt.CacheStamp()	20
ubelt.modname_to_modpath()	17
ubelt.highlight_code()	17
ubelt.import_module_from_name()	16
ubelt.find_exe()	14
ubelt.inject_method()	13
ubelt.shrinkuser()	11
ubelt.AutoDict()	9
ubelt.argmin()	9
ubelt.find_path()	7
ubelt.download()	5
ubelt.sorted_keys()	5
ubelt.CaptureStdout()	4
ubelt.split_modpath()	4
ubelt.modpath_to_modname()	4
ubelt.orderedset()	4
ubelt.userhome()	3
ubelt.argunique()	3
ubelt.AutoOrderedDict()	2
ubelt.unique_flags()	2
ubelt.boolmask()	1

```
usage_stats = {
    'mean': 73.12658,
```

(continues on next page)

(continued from previous page)

```
    'std': 170.95476,
    'min': 1.0,
    'max': 1478.0,
    'med': 27.0,
    'sum': 5777,
    'shape': (79,),
}
```

1.1 ubelt.orderedset

ubelt.OrderedSet() *ubelt.oset()*

1.2 ubelt.progiter

ubelt.ProgIter()

1.3 ubelt.timerit

ubelt.Timer() *ubelt.Timerit()*

1.4 ubelt.util_arg

ubelt.PY2() ubelt.string_types() *ubelt.argval()* *ubelt.argflag()*

1.5 ubelt.util_cache

ubelt.Cacher() *ubelt.CacheStamp()*

1.6 ubelt.util_cmd

ubelt.cmd()

1.7 ubelt.util_colors

ubelt.NO_COLOR() *ubelt.highlight_code()* *ubelt.color_text()*

1.8 ubelt.util_const

ubelt.NoParam()

1.9 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()*

1.10 ubelt.util_download

ubelt.download() *ubelt.grabdata()*

1.11 ubelt.util_format

ubelt.PY2() ubelt.iteritems() ubelt.string_types() ubelt.text_type() *ubelt.repr2()* *ubelt.FormatterExtensions()*

1.12 ubelt.util_func

ubelt.identity() *ubelt.inject_method()*

1.13 ubelt.util_hash

ubelt.hash_data() *ubelt.hash_file()*

1.14 ubelt.util_import

ubelt.split_modpath() *ubelt.modname_to_modpath()* *ubelt.modpath_to_modname()* *ubelt.import_module_from_name()* *ubelt.import_module_from_path()*

1.15 ubelt.util_io

ubelt.readfrom() *ubelt.writeto()* *ubelt.touch()* *ubelt.delete()*

1.16 ubelt.util_links

ubelt.PY2() *ubelt.symlink()*

1.17 ubelt.util_list

ubelt.PY2() ubelt.string_types() *ubelt.chunks()* *ubelt.iterable()* *ubelt.take()* *ubelt.compress()* *ubelt.flatten()* *ubelt.unique()* *ubelt.argunique()* *ubelt.unique_flags()* *ubelt.boolmask()* *ubelt.iter_window()* *ubelt.allsame()* *ubelt.argsort()* *ubelt.argmax()* *ubelt.argmin()* *ubelt.peek()*

1.18 ubelt.util_memoize

ubelt.PY2() *ubelt.memoize()* *ubelt.memoize_method()* *ubelt.memoize_property()*

1.19 ubelt.util_mixins

ubelt.NiceRepr()

1.20 ubelt.util_path

ubelt.TempDir() *ubelt.augpath()* *ubelt.shrinkuser()* *ubelt.userhome()* *ubelt.ensuredir()* *ubelt.expandpath()*

1.21 ubelt.util_platform

ubelt.WIN32() ubelt.LINUX() ubelt.DARWIN() ubelt.POSIX() ubelt.PY2() ubelt.string_types() *ubelt.platform_data_dir()* *ubelt.platform_config_dir()* *ubelt.platform_cache_dir()* *ubelt.get_app_data_dir()* *ubelt.ensure_app_data_dir()* *ubelt.get_app_config_dir()* *ubelt.ensure_app_config_dir()* *ubelt.get_app_cache_dir()* *ubelt.ensure_app_cache_dir()* *ubelt.find_exe()* *ubelt.find_path()*

1.22 ubelt.util_str

ubelt.indent() *ubelt.codeblock()* *ubelt.paragraph()* *ubelt.hzcat()* *ubelt.ensure_unicode()*

1.23 ubelt.util_stream

ubelt.PY2() *ubelt.TeeStringIO()* *ubelt.CaptureStream()* *ubelt.CaptureStdout()*

1.24 ubelt.util_time

ubelt.timestamp()

1.24.1 ubelt

1.24.1.1 ubelt package

1.24.1.1.1 Submodules

1.24.1.1.1.1 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*

1.24.1.1.1.2 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 inferface is most common.

ProgIter was originally developed independantly 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_**. The `tqdm` library may be more appropriate in some cases. *The main advantage of ``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 desireable 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
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*, ***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.
- **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()
```

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.

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 progres 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

1.24.1.1.1.3 ubelt.timerit module

First, *Timer* is a context manager that times a block of indented code. Also has *tic* and *toc* methods for a more matlab like feel.

Next, *Timerit* is an alternative to the builtin timeit module. I think its better at least, maybe Tim Peters can show me otherwise. Perhaps there's a reason it has to work on strings and can't be placed around existing code like a with statement.

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> #
>>> # The Timerit class allows for robust benchmarking based
>>> # It can be used in normal scripts by simply adjusting the indentation
>>> import math
>>> for timer in Timerit(num=12, verbose=3):
>>>     with timer:
>>>         math.factorial(100)
Timing for: 200 loops, best of 3
Timed for: 200 loops, best of 3
    body took: 331.840 µs
    time per loop: best=1.569 µs, mean=1.615 ± 0.0 µs
```

```
>>> # xdoctest: +SKIP
>>> # In Contrast, timeit is similar, but not having to worry about setup
>>> # and inputing the program as a string, is nice.
>>> import timeit
>>> timeit.timeit(stmt='math.factorial(100)', setup='import math')
1.12695...
```

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> #
>>> # The Timer class can also be useful for quick checks
>>> #
>>> import math
>>> timer = Timer('Timer demo!', verbose=1)
>>> x = 100000  # the input for example output
>>> x = 10      # the input for test speed considerations
>>> with timer:
>>>     math.factorial(x)
tic('Timer demo!')
...toc('Timer demo!')=0.1959s
```

class ubelt.timerit.**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

class ubelt.timerit.**Timerit**(*num=1*, *label=None*, *bestof=3*, *unit=None*, *verbose=None*)
Bases: object

Reports the average time to run a block of code.

Unlike *%timeit*, *Timerit* can handle multiline blocks of code. It runs inline, and doesn't depend on magic or strings. Just indent your code and place in a Timerit block. See https://github.com/Erotemic/vimtk for vim functions that will insert one of these in for you (ok that part is a little magic).

Parameters

- **num** (*int, default=1*) – number of times to run the loop
- **label** (*str, default=None*) – identifier for printing
- **bestof** (*int, default=3*) – takes the max over this number of trials
- **unit** (*str*) – what units time is reported in
- **verbose** (*int*) – verbosity flag, defaults to True if label is given

Variables

- **– labeled measurements taken by this object** (*measures*) –
- **– ranked measurements** (*rankings*) –

Example

```
>>> import math
>>> num = 3
>>> t1 = Timerit(num, label='factorial', verbose=1)
>>> for timer in t1:
>>>     # <write untimed setup code here> this example has no setup
>>>     with timer:
>>>         # <write code to time here> for example...
>>>         math.factorial(100)
Timed best=..., mean=... for factorial
>>> # <you can now access Timerit attributes>
>>> assert t1.total_time > 0
>>> assert t1.n_loops == t1.num
>>> assert t1.n_loops == num
```

Example

```
>>> # xdoc: +IGNORE_WANT
>>> import math
>>> num = 4
>>> # If the timer object is unused, time will still be recorded,
>>> # but with less precision.
>>> for _ in Timerit(num, 'concise', bestof=2, verbose=2):
>>>     math.factorial(100)
Timed concise for: 4 loops, best of 2
    time per loop: best=1.637 µs, mean=1.935 ± 0.3 µs
>>> # Using the timer object results in the most precise timings
>>> for timer in Timerit(num, 'precise', bestof=2, verbose=3):
>>>     with timer: math.factorial(100)
Timed precise for: 4 loops, best of 2
    body took: 8.696 µs
    time per loop: best=1.754 µs, mean=1.821 ± 0.1 µs
```

reset (*label=None*, *measures=False*)
clears all measurements, allowing the object to be reused

Parameters

- **label** (*str, optional*) – change the label if specified
- **measures** (*bool, default=False*) – if True reset measures

Example

```
>>> import math
>>> ti = Timerit(num=10, unit='us', verbose=True)
>>> _ = ti.reset(label='10!').call(math.factorial, 10)
Timed best=...s, mean=...s for 10!
>>> _ = ti.reset(label='20!').call(math.factorial, 20)
Timed best=...s, mean=...s for 20!
>>> _ = ti.reset().call(math.factorial, 20)
Timed best=...s, mean=...s for 20!
>>> _ = ti.reset(measures=True).call(math.factorial, 20)
```

call (*func*, **args*, ***kwargs*)
Alternative way to time a simple function call using condensed syntax.

Returns

Use *min*, or *mean* to get a scalar. Use *print* to output a report to stdout.

Return type self (*Timerit*)

Example

```
>>> import math
>>> time = Timerit(num=10).call(math.factorial, 50).min()
>>> assert time > 0
```

property rankings
Orders each list of measurements by ascending time

Example

```
>>> import math
>>> ti = Timerit(num=1)
>>> _ = ti.reset('a').call(math.factorial, 5)
>>> _ = ti.reset('b').call(math.factorial, 10)
>>> _ = ti.reset('c').call(math.factorial, 20)
>>> ti.rankings
>>> ti.consistency
```

property consistency
” Take the hamming distance between the preference profiles to as a measure of consistency.

min ()
The best time overall.

This is typically the best metric to consider when evaluating the execution time of a function. To understand why consider this quote from the docs of the original timeit module:

‘” In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python’s speed, but by other processes interfering with your timing accuracy. So the min() of the result is probably the only number you should be interested in. ‘”

Returns minimum measured seconds over all trials

Return type float

Example

```
>>> import math
>>> self = Timerit(num=10, verbose=0)
>>> self.call(math.factorial, 50)
>>> assert self.min() > 0
```

mean()

The mean of the best results of each trial.

Returns mean of measured seconds

Return type float

Note: This is typically less informative than simply looking at the min. It is recommended to use min as the expectation value rather than mean in most cases.

Example

```
>>> import math
>>> self = Timerit(num=10, verbose=0)
>>> self.call(math.factorial, 50)
>>> assert self.mean() > 0
```

std()

The standard deviation of the best results of each trial.

Returns standard deviation of measured seconds

Return type float

Note: As mentioned in the timeit source code, the standard deviation is not often useful. Typically the minimum value is most informative.

Example

```
>>> import math
>>> self = Timerit(num=10, verbose=1)
>>> self.call(math.factorial, 50)
>>> assert self.std() >= 0
```

report(*verbose=1*)

Creates a human readable report

Parameters **verbose** (*int*) – verbosity level. Either 1, 2, or 3.

Returns the report

Return type str

SeeAlso: *Timerit.print()*

Example

```
>>> import math
>>> ti = Timerit(num=1).call(math.factorial, 5)
>>> print(ti.report(verbose=1))
Timed best=...s, mean=...s
```

print(*verbose=1*)

Prints human readable report using the print function

Parameters **verbose** (*int*) – verbosity level

SeeAlso: *Timerit.report()*

Example

```
>>> import math
>>> Timerit(num=10).call(math.factorial, 50).print(verbose=1)
Timed best=...s, mean=...s
>>> Timerit(num=10).call(math.factorial, 50).print(verbose=2)
Timed for: 10 loops, best of 3
    time per loop: best=...s, mean=...s
>>> Timerit(num=10).call(math.factorial, 50).print(verbose=3)
Timed for: 10 loops, best of 3
    body took: ...
    time per loop: best=...s, mean=...s
```

1.24.1.1.1.4 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 conjuction 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** (*object, default=NoParam*) – a value to return if not specified.
- **argv** (*Optional[list], 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

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
```

1.24.1.1.1.5 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 ever achieved. These four lines implement the following necessary and sufficient steps for general robust on-disk caching.

1. Defining the cache dependenies
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
>>> # Defines a cache name and dependencies, note the use of `ub.hash_data`.
>>> cacher = ub.Cacher('name', cfgstr=ub.hash_data('dependencies'))      #␣
↪boilerplate:1
>>> # Calling tryload will return your data on a hit and None on a miss
>>> data = cacher.tryload()                                             #␣
↪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 (pickle is used by default)
>>>     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', cfgstr=ub.hash_data('dependencies'))   # boilerplate:1
>>> def func():                                              # boilerplate:2
>>>     data = 'mydata'
>>>     return data                                          # boilerplate:3
>>> data = func()                                            # boilerplate:4
```

```
>>> cacher = ub.Cacher('name', cfgstr=ub.hash_data('dependencies'))   # boilerplate:1
>>> data = cacher.tryload()                                          # 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 simpler tracebacks, explicit procedures, and makes it easier to use breakpoint debugging (because there is no closure scope).

While *Cacher* is used to store simple 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

```
>>> def compute_many_files(dpath):
...     for i in range(0):
...         fpath = '{}/file{}.txt'.format(dpath, i)
...         open(fpath).write('foo' + str(i))
>>> #
>>> import ubelt as ub
>>> dpath = ub.ensure_app_cache_dir('ubelt/demo/cache')
>>> # You must specify a directory, unlike in Cacher where it is optional
>>> self = ub.CacheStamp('name', dpath=dpath, cfgstr='dependencies')
>>> 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()
```

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*)

Bases: `object`

Cacher designed to be quickly integrated into existing scripts.

A dependency string can be specified, which will invalidate the cache if it changes to an unseen value. The location

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]*) – Indicate dependencies of this cache. If the dependencies change, then the cache is recomputed. New in version 0.8.9, replaces *cfgstr*.
- **dpath** (*PathLike*) – Specifies where to save the cache. If unspecified, Cacher defaults to an application resource dir as given by appname.
- **appname** (*str, default='ubelt'*) – Application name Specifies a folder in the application resource directory where to cache the data if `dpath` is not specified.
- **ext** (*str, default='.pkl'*) – File extension for the cache format
- **meta** (*object*) – Metadata that is also saved with the `cfgstr`. This can be useful to indicate how the `cfgstr` was constructed.
- **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** (*func*) – Overloads the print function. Useful for sending output to loggers (e.g. logging.info, tqdm.tqdm.write, ...)
- **hasher** (*str*) – Type of hashing algorithm to use if `cfgstr` needs to be condensed to less than 49 characters.

- **protocol** (*int, default=2*) – Protocol version used by pickle. If python 2 compatibility is not required, then it is better to use protocol 4.
- **cfgstr** (*str*) – Deprecated in favor of depends. Indicates the state. Either this string or a hash of this string will be used to identify the cache. A cfgstr should always be reasonably readable, thus it is good practice to hash extremely detailed cfgstrs to a reasonable readable level. Use meta to store make original details persist.

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()
>>> 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

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, optional*) – overrides the instance-level cfgstr

Returns 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', cfgstr='cfg1')
>>> self.get_fpath()
>>> self = Cacher('test_cacher3', cfgstr='cfg1' * 32)
>>> self.get_fpath()
```

exists(*cfgstr=None*)
Check to see if the cache exists

Parameters **cfgstr** (*str, optional*) – 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()
>>> cacher = Cacher('versioned_data_v2', depends='1')
>>> cacher.ensure(lambda: 'data1')
>>> known_fpaths.add(cacher.get_fpath())
>>> cacher = Cacher('versioned_data_v2', depends='2')
>>> 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')
>>> 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, optional*) – 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, optional*) – 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, optional*) – 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(cfgstr)*.

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, optional*) – overrides the instance-level cfgstr

Example

```
>>> from ubelt.util_cache import *  # NOQA
>>> # Normal functioning
>>> cfgstr = 'long-cfg' * 32
>>> cacher = Cacher('test_enabled_save', cfgstr)
>>> cacher.save('data')
>>> assert exists(cacher.get_fpath()), 'should be enabeled'
>>> 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'
>>> cfgstr = 'func params'
>>> cacher = Cacher(fname, cfgstr)
>>> 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*)

Bases: `object`

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

Writes a file that marks that a procedure has been done by writing a "stamp" file to disk. Removing the stamp file will force recomputation. However, removing or changing the result of the computation may not trigger recomputation unless specific handling is done or the expected "product" of the computation is a file and registered with the stamper. If hasher is None, we only check if the product exists, and we ignore its hash, otherwise it checks that the hash of the product is the same.

Parameters

- **fname** (*str*) – Name of the stamp file
- **cfgstr** (*str*) – Configuration associated with the stamped computation. A common pattern is to call *ubelt.hash_data()* on a dependency list.

  Deprecated in favor of depends. Indicates the state. Either this string or a hash of this string will be used to identify the cache. A cfgstr should always be reasonably readable, thus it is good practice to hash extremely detailed cfgstrs to a reasonable readable level. Use meta to store make original details persist.
- **dpath** (*PathLike*) – Where to store the cached stamp file
- **product** (*PathLike or Sequence[PathLike], optional*) – 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. Defaults to sha1.
- **verbose** (*bool, default=None*) – Passed to internal ub.Cacher object
- **enabled** (*bool, default=True*) – if False, expired always returns True
- **depends** (*str | List[str]*) – 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*) – 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.

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> # Stamp the computation of expensive-to-compute.txt
>>> dpath = ub.ensure_app_cache_dir('ubelt', 'test-cache-stamp')
>>> ub.delete(dpath)
>>> ub.ensuredir(dpath)
>>> product = join(dpath, 'expensive-to-compute.txt')
>>> self = CacheStamp('somedata', cfgstr='someconfig', dpath=dpath,
>>>                   product=product, hasher=None)
>>> self.hasher = None
>>> if self.expired():
>>>     ub.writeto(product, 'very expensive')
>>>     self.renew()
>>> assert not self.expired()
>>> # corrupting the output will not expire in non-robust mode
>>> ub.writeto(product, 'corrupted')
>>> assert not self.expired()
>>> self.hasher = 'sha1'
>>> # but it will expire if we are in robust mode
>>> assert self.expired()
>>> # deleting the product will cause expiration in any mode
>>> self.hasher = None
>>> ub.delete(product)
>>> assert self.expired()
```

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

Check to see if a previously existing stamp is still valid and if the expected result of that computation still exists.

Parameters

- **cfgstr** (*str, optional*) – overrides the instance-level cfgstr
- **product** (*PathLike or Sequence[PathLike], optional*) – override the default product if specified

Returns True if the stamp is invalid or does not exist.

Return type bool

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

1.24.1.1.1.6 ubelt.util_cmd module

This module exposes the *ubelt.cmd()* command, which provides a simple means for interacting with the commandline. While this does use `subprocess.Popen` under the hood, the key draw of *ubelt.cmd()* is that you can capture stdout/stderr in your program while simultaneously printing it to the terminal in real time.

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 `detatch=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*, ***kwargs*)

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 or Sequence*) – 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, optional*) – 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** (*PathLike, optional*) – path to run command
- **env** (*str, optional*) – environment passed to Popen
- **tee_backend** (*str, optional*) – 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.
- ****kwargs** – only used to support deprecated arguments

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

Notes

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[3] for a potential cross-platform shlex solution for windows.

CommandLine: python -m ubelt.util_cmd cmd python -c "import ubelt as ub; ub.cmd('ping localhost -c 2', verbose=2)"

References

Example

```
>>> info = 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

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

Example

```
>>> # windows echo will output extra single quotes
>>> info = 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
>>> info = cmd('echo str&&echo shell', verbose=0, shell=True)
>>> assert info['out'].strip() == 'str' + chr(10) + 'shell'
```

[3] https://stackoverflow.com/questions/33560364/python-windows-parsing-command-lines-with-shlex

Example

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

Example

```
>>> import pytest
>>> info = cmd('echo hi', check=True)
>>> import subprocess
>>> with pytest.raises(subprocess.CalledProcessError):
>>>     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)
>>> info1 = ub.cmd(('touch', fpath1), detach=True)
>>> info2 = ub.cmd('echo writing2 > ' + fpath2, shell=True, detach=True)
>>> while not exists(fpath1):
...     pass
>>> while not exists(fpath2):
...     pass
>>> assert ub.readfrom(fpath1) == ''
>>> assert ub.readfrom(fpath2).strip() == 'writing2'
>>> info1['proc'].wait()
>>> info2['proc'].wait()
```

1.24.1.1.1.7 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 programing 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 complient with[1].

[1] https://no-color.org/

References

Notes

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)
```

(continues on next page)

(continued from previous page)

```
>>>     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'
```

1.24.1.1.1.8 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
```

1.24.1.1.1.9 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_subset()* 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[A]*) – full sequence
- **items2** (*Iterable[B]*) – 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[A, B]

Example

```
>>> assert dzip([1, 2, 3], [4]) == {1: 4, 2: 4, 3: 4}
>>> assert dzip([1, 2, 3], [4, 4, 4]) == {1: 4, 2: 4, 3: 4}
>>> assert 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')
>>> 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[A, B]*) – superset dictionary
- **keys** (*Iterable[A]*) – keys to take from `dict_`
- **default** (*object, optional*) – if specified uses default if keys are missing
- **cls** (*type, default=OrderedDict*) – type of the returned dictionary.

Returns subset dictionary

Return type cls[A, B]

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*)

Combines the disjoint keys in multiple dictionaries. For intersecting keys, dictionaries towards the end of the sequence are given precedence.

Parameters ***args** – a sequence of dictionaries

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

Return type Dict | OrderedDict

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

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

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

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

Parameters ***args** – a sequence of dictionaries (or sets of keys)

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

Return type Dict | OrderedDict

Notes

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

```
>>> dict_isect({'a': 1, 'b': 2, 'c': 3}, ['a', 'c', 'd'])
{'a': 1, 'c': 3}
```

Example

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

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

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

Parameters ***args** – a sequence of dictionaries (or sets of keys)

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

Return type Dict | OrderedDict

Todo:

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

Example

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

List[int]]: maps each duplicate item to the indices at which it appears

Return type dict[T

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[A]*) – a list of items to group
- **key** (*Iterable[B] | Callable[[A], B]*) – 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[B, List[A]]

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[A, B]*) – 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[B, A] | Dict[B, Set[A]]

Notes

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[[A], C] | Mapping[A, C]*) – a function or indexable object
- **dict_** (*Dict[A, B]*) – a dictionary

Returns transformed dictionary

Return type Dict[C, B]

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

Example

```
>>> dict_ = {'a': [1, 2, 3], 'b': []}
>>> func = ord
>>> newdict = map_keys(func, dict_)
>>> print(newdict)
>>> assert newdict == {97: [1, 2, 3], 98: []}
>>> dict_ = {0: [1, 2, 3], 1: []}
>>> func = ['a', 'b']
>>> newdict = 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[[B], C] | Mapping[B, C]*) – a function or indexable object
- **dict_** (*Dict[A, B]*) – a dictionary

Returns transformed dictionary

Return type Dict[A, C]

Example

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

Example

```
>>> # Can also use an indexable as ``func``
>>> dict_ = {'a': 0, 'b': 1}
>>> func = [42, 21]
>>> newdict = 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[A, B]*) – dictionary to sort. The keys must be of comparable types.
- **key** (*Callable[[A], Any], optional*) – 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[A, B]

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[A, B]*) – dictionary to sort. The values must be of comparable types.
- **key** (*Callable[[B], Any], optional*) – 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[A, B]

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

1.24.1.1.1.10 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*)

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** (*PathLike | io.BytesIOtringIO*) – 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** (*PathLike*) – where to download the file. If unspecified *appname* is used to determine this. Mutually exclusive with fpath.

- **fname** (*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.

Returns fpath - path to the downloaded file.

Return type PathLike

Raises

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

Notes

Based largely on code in pytorch[4] with modifications influenced by other resources[1,2,3].

References

Todo:

- [ ] fine-grained control of progress

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
```

[4] https://github.com/pytorch/pytorch/blob/2787f1d8edbd4aadd4a8680d204341a1d7112e2d/torch/hub.py#L347
[1] http://blog.moleculea.com/2012/10/04/urlretrieve-progres-indicator/
[2] http://stackoverflow.com/questions/15644964/python-progress-bar-and-downloads
[3] http://stackoverflow.com/questions/16694907/how-to-download-large-file-in-python-with-requests-py

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import ubelt as ub
>>> import io
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> file = io.BytesIO()
>>> fpath = 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)
>>> # test download from girder
>>> import pytest
>>> import ubelt as ub
>>> 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.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 to the file to download
- **fpath** (*PathLike*) – The full path to download the file to. If unspecified, the arguments *dpath* and *fname* are used to determine this.
- **dpath** (*PathLike*) – where to download the file. If unspecified *appname* is used to determine this. Mutually exclusive with fpath.
- **fname** (*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.
- ****download_kw** – additional kwargs to pass to ub.download

Returns fpath - path to downloaded or cached file.

Return type 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
>>> fname = 'foo.bar'
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> prefix1 = '944389a39dfb8fa9'
>>> fpath = ub.grabdata(url, fname=fname, hash_prefix=prefix1)
>>> stamp_fpath = fpath + '.sha512.hash'
>>> assert ub.readfrom(stamp_fpath) == 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
>>> open(stamp_fpath, 'w').write('corrupt-stamp')
>>> fpath = ub.grabdata(url, fname=fname, hash_prefix=prefix1)
>>> assert ub.readfrom(stamp_fpath) == prefix1
>>> #
>>> # Check that a redownload occurs when the stamp is removed
>>> ub.delete(stamp_fpath)
>>> open(fpath, 'w').write('corrupt-data')
>>> assert not ub.hash_file(fpath, base='hex', hasher='sha512').
↪startswith(prefix1)
```

(continues on next page)

(continued from previous page)

```
>>> 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'
>>> fpath = ub.grabdata(url2, fname=fname, hash_prefix=prefix2)
>>> assert ub.readfrom(stamp_fpath) == prefix2
```

1.24.1.1.1.11 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 normaly 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)}
```

ubelt.util_format.**iteritems**(*d*, ***kw*)

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 very configurable. By default it aims to produce strings that are executable and consistent between Python versions. This makes them great for doctests.

Notes

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

Parameters

- **data** (*object*) – an arbitrary python object
- ****kwargs** – see "the Kwargs" section

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, ...)`.

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 determenistic 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

Notes

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

Example

```
>>> from ubelt.util_format import *
>>> 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 = 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}],
    '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 = repr2(dict_, nl=3, precision=2); print(result)
>>> result = repr2(dict_, nl=2, precision=2); print(result)
>>> result = repr2(dict_, nl=1, precision=2, itemsep='', explicit=True);␣
↪print(result)
>>> result = repr2(dict_, nl=1, precision=2, nobr=1, itemsep='', explicit=True);␣
↪print(result)
>>> result = repr2(dict_, nl=3, precision=2, cbr=True); print(result)
>>> result = repr2(dict_, nl=3, precision=2, si=True); print(result)
>>> result = repr2(dict_, nl=3, sort=True); print(result)
>>> result = repr2(dict_, nl=3, sort=False, trailing_sep=False); print(result)
>>> result = repr2(dict_, nl=3, sort=False, trailing_sep=False, nobr=True);␣
↪print(result)
```

Example

```
>>> from ubelt.util_format import *
>>> 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 = repr2(dict_, nl=6, precision=2, cbr=1)
>>> print('---')
>>> print(result)
>>> result = repr2(dict_, nl=-1, precision=2)
>>> print('---')
>>> print(result)
```

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 explicilty 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.

1.24.1.1.1.12 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()` 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[[T, ...], 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

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

1.24.1.1.1.13 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 *:func: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 *:func:hash_file*

The `ub.hash_data()` function recursively hashes most builtin python data structures.

The `ub.hash_file()` function hashes data on disk. Both of the aformentioned 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, convert=False, hasher='sha512')
2ff39d0ecbf6ecc740ca7d...
```

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.

`ubelt.util_hash.`**`hash_data`**(*data*, *hasher=NoParam*, *base=NoParam*, *types=False*, *hashlen=NoParam*, *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 | hashlib.HASH, 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.
- **hashlen** (*int*) – Maximum number of symbols in the returned hash. If not specified, all are returned. DEPRECATED. Use slice syntax instead.
- **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.

Notes

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

Notes

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*, *hashlen=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 dependant on blocksize.
- **maxbytes** (*int | None*) – if specified, only hash the leading *maxbytes* of data in the file.
- **hasher** (*str | hashlib.HASH, 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
- **hashlen** (*int*) – maximum number of symbols in the returned hash. If not specified, all are returned. DEPRECATED. DO NOT USE.
- **base** (*List[str] | str, default='hex'*) – list of symbols or shorthand key. Valid keys are 'abc', 'hex', and 'dec'.

Notes

For better hashes keep stride = 1 For faster hashes set stride > 1 blocksize matters when stride > 1

References

http://stackoverflow.com/questions/3431825/md5-checksum-of-a-file http://stackoverflow.com/questions/5001893/when-to-use-sha-1-vs-sha-2

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> fpath = join(ub.ensure_app_cache_dir('ubelt'), 'tmp.txt')
>>> ub.writeto(fpath, 'foobar')
>>> print(ub.hash_file(fpath, hasher='sha1', base='hex'))
8843d7f92416211de9ebb963ff4ce28125932878
```

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> fpath = join(ub.ensure_app_cache_dir('ubelt'), 'tmp.txt')
>>> ub.writeto(fpath, 'foobar')
>>> print(ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=1000))
8843d7f92416211de9ebb963ff4ce28125932878
```

```
>>> # We have the ability to only hash at most ``maxbytes`` in a file
>>> ub.writeto(fpath, '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 dependant 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)
>>> assert h1 != h2 != h3
>>> assert h1 == h0
>>> assert h2 == h4
```

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> fpath = ub.touch(join(ub.ensure_app_cache_dir('ubelt'), 'empty_file'))
>>> # Test that the output is the same as sha1sum
>>> 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)
```

1.24.1.1.1.14 ubelt.util_import module

Expose functions to simplify importing from module names and paths.

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

The `ub.import_module_from_name()` works similarly.

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

The `ub.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

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 (__name__)

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

Returns module

Return type module

Example

```
>>> # test with modules that wont be imported in normal circumstances
>>> # todo write a test where we gaurentee 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** (*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 module

References

https://stackoverflow.com/questions/67631/import-module-given-path

Notes

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.

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
>>> 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 module.__name__ == os.path.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/')
```

1.24.1.1.1.15 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 *open(<fpath>).read()* 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: You probably should 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: It turns out that *open(<fpath>).write(<to_write>)* does not work in pypy. See *https://pypy.org/compat.html*. This is a strong argument for keeping this function.

Example

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

(continues on next page)

(continued from previous page)

```
>>> read_ = ub.readfrom(fpath)
>>> print('read_    = ' + read_)
>>> print('to_write = ' + to_write)
>>> assert read_ == to_write
```

Example

```
>>> import ubelt as ub
>>> 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']
>>> 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
```

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** (*file*) – 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

https://stackoverflow.com/questions/1158076/implement-touch-using-python

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. https://github.com/hsoft/send2trash

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)
```

1.24.1.1.1.16 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. Obviously, any POSIX system work 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 symbolic link.

This will work on linux or windows, however windows does have some corner cases. For more details see notes in `ubelt._win32_links`.

Parameters

- **path** (*PathLike*) – path to real file or directory
- **link_path** (*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 PathLike

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'))
```

(continues on next page)

(continued from previous page)

```
>>> 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)
```

1.24.1.1.1.17 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()*.

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

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

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

References

http://stackoverflow.com/questions/434287/iterate-over-a-list-in-chunks

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))) == 2
>>> assert len(list(ub.chunks([], 2, None, 'none'))) == 0
>>> assert len(list(ub.chunks([], 2, None, 'cycle'))) == 0
>>> assert len(list(ub.chunks([], 2, None, 'replicate'))) == 0
```

Example

```
>>> 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

```
>>> 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))
```

static noborder(*items*, *chunksize*)

static cycle(*items*, *chunksize*)

static replicate(*items*, *chunksize*)

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

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

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

Selects a subset of a list based on a list of indices.

This is similar to `np.take`, but pure python. This also supports specifying a default element if `items` is an iterable of dictionaries.

Parameters

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

Yeilds: V: a selected item within the list

SeeAlso: `ub.dict_subset()`

Notes

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

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

Selects from `items` where the corresponding value in `flags` is True. This is similar to `numpy.compress()`.

This is actually a simple alias for `itertools.compress()`.

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]

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.

This is simply an alias for `itertools.chain.from_iterable()`.

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

Returns flattened items

Return type Iterable[Any]

Example

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

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

Generates unique items in the order they appear.

Parameters

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

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

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
>>> import six
>>> items = ['A', 'a', 'b', 'B', 'C', 'c', 'D', 'e', 'D', 'E']
>>> unique_items = list(ub.unique(items, key=six.text_type.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.**argunique**(*items*, *key=None*)
Returns indices corresponding to the first instance of each unique item.

Parameters

- **items** (*Sequence[V]*) – indexable collection of items
- **key** (*Callable[[V], 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

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

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*) – indexable collection of items
- **key** (*Callable[[V], Any], 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 = unique_flags(items)
>>> assert flags == [True, True, True, False, False, True, False]
>>> flags = unique_flags(items, key=lambda x: x % 2 == 0)
>>> assert flags == [True, False, True, False, False, False, False]
```

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]
```

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 overlaping windows in a sequence

Return type Iterable[T]

Example

```
>>> iterable = [1, 2, 3, 4, 5, 6]
>>> size, step, wrap = 3, 1, True
>>> window_iter = 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

```
>>> iterable = [1, 2, 3, 4, 5, 6]
>>> size, step, wrap = 3, 2, True
>>> window_iter = 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

```
>>> iterable = [1, 2, 3, 4, 5, 6]
>>> size, step, wrap = 3, 2, False
>>> window_iter = 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

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

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

Determine if all items in a sequence are the same

Parameters

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

Returns True if all items are equal, otherwise False

Return type bool

Example

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

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

- **reverse** (*bool, default=False*) – if True returns in descending order

Returns indices - list of indices such that sorts the indexable

Return type List[int]

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.**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[B] | Mapping[A, B]*) – indexable to sort by
- **key** (*Callable[[A], B], default=None*) – customizes the ordering of the indexable

Returns the index of the item with the maximum value.

Return type int

Example

```
>>> assert argmax({'a': 3, 'b': 2, 'c': 100}) == 'c'
>>> assert argmax(['a', 'c', 'b', 'z', 'f']) == 3
>>> assert argmax([[0, 1], [2, 3, 4], [5]], key=len) == 1
>>> assert argmax({'a': 3, 'b': 2, 3: 100, 4: 4}) == 3
>>> assert 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[B] | Mapping[A, B]*) – indexable to sort by
- **key** (*Callable[[A], B], default=None*) – customizes the ordering of the indexable

Returns the index of the item with the minimum value.

Return type int

Example

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

ubelt.util_list.**peek**(*iterable*)

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** (*List[T]*) – an iterable

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

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
```

1.24.1.1.1.18 ubelt.util_memoize module

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

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)
>>> 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

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

Returns memoized wrapper

Return type Callable

References

https://wiki.python.org/moin/PythonDecoratorLibrary#Memoize

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

http://code.activestate.com/recipes/577452-a-memoize-decorator-for-instance-methods/

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'
```

(continues on next page)

(continued from previous page)

```
>>> 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.

Notes

implementation is a modified version of [1].

References

..[1] https://github.com/estebistec/python-memoized-property

Example

```
>>> class C(object):
...     load_name_count = 0
...     @memoize_property
...     def name(self):
...         "name's docstring"
...         self.load_name_count += 1
...         return "the name"
...     @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
```

(continues on next page)

(continued from previous page)

```
'the name'
>>> c.load_name_count
1
>>> c.name
'the name'
>>> c.load_name_count
1
>>> c.another_name
```

1.24.1.1.1.19 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')
>>> # 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])
```

(continues on next page)

(continued from previous page)

```
>>> 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 ...>
```

1.24.1.1.1.20 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.

class ubelt.util_path.**TempDir**

Bases: object

Context for creating and cleaning up temporary directories.

Example

```
>>> with TempDir() as self:
>>>     dpath = self.dpath
>>>     assert exists(dpath)
>>> assert not exists(dpath)
```

Example

```
>>> 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*, *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, default=''*) – placed between the basename and extension
- **prefix** (*str, default=''*) – placed in front of the basename
- **ext** (*str, default=None*) – if specified, replaces the extension
- **base** (*str, default=None*) – if specified, replaces the basename without extension
- **dpath** (*str | PathLike, default=None*) – if specified, replaces the specified "relative" directory, which by default is the parent directory.
- **relative** (*str | PathLike, default=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, default=False*) – 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

```
>>> 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'
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

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

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

Parameters

- **path** (*str | PathLike*) – path in system file structure
- **home** (*str, default='~'*) – 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 tilde

Return type str

Example

```
>>> 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, default=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

```
>>> import getpass
>>> username = getpass.getuser()
>>> assert userhome() == expanduser('~')
>>> assert userhome(username) == expanduser('~')
```

ubelt.util_path.**ensuredir**(*dpath*, *mode=1023*, *verbose=None*, *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, default=0o1777*) – octal mode of directory
- **verbose** (*int, default=0*) – verbosity
- **recreate** (*bool, default=False*) – 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

Notes

This function is not thread-safe in Python2

Example

```
>>> from ubelt.util_platform 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.

Less aggressive than truepath. Only expands environs and tilde. Does not change relative paths to absolute paths.

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

Returns expanded path

Return type str

Example

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

1.24.1.1.1.21 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[1]. This is similar to the more focused `appdirs` module[5]. In the future ubelt may directly use `appdirs`.

Notes

Table mapping the type of directory to the system default environment variable. Inspired by[2],[3], and[4].

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

References

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

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 cahce dir used by the current operating system

Return type str

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.**get_app_data_dir**(*appname*, **args*)
Returns a writable directory for an application. This should be used for temporary deletable data.

Parameters

[1] https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html
[5] https://github.com/ActiveState/appdirs
[2] https://stackoverflow.com/questions/43853548/xdg-windows
[3] https://stackoverflow.com/questions/11113974/cross-plat-path
[4] https://github.com/harawata/appdirs#supported-directories

- **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.**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_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.**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.**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.**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.**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], 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+.

Notes

This is essentially the `which` UNIX command

References

https://stackoverflow.com/questions/377017/test-if-executable-exists-in-python/377028#377028 https://docs.python.org/dev/library/shutil.html#shutil.which

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

```
>>> assert not find_exe('noexist', multi=False)
>>> assert find_exe('ping', multi=False)
>>> assert not find_exe('noexist', multi=True)
>>> assert find_exe('ping', 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

- **fname** (*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`

Notes

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

Notes

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('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
```

1.24.1.1.1.22 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

```
>>> from ubelt.util_str import *  # NOQA
>>> NL = chr(10)  # newline character
>>> text = 'Lorem ipsum' + NL + 'dolor sit amet'
>>> prefix = '    '
>>> result = 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

```
>>> from ubelt.util_str import *  # NOQA
>>> # Simulate an indented part of code
>>> if True:
>>>     # notice the indentation on this will be normal
>>>     codeblock_version = 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

```
>>> from ubelt.util_str import *  # NOQA
>>> text = (
>>>     '''
>>>     Lorem ipsum dolor sit amet, consectetur adipiscing
>>>     elit, sed do eiusmod tempor incididunt ut labore et
>>>     dolore magna aliqua.
>>>     ''')
>>> out = 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:

```
>>> from ubelt.util_str import *
>>> 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)

References

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')
```

1.24.1.1.1.23 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

```
>>> redirect = io.StringIO()
>>> self = TeeStringIO(redirect)
```

isatty()

Returns true of the redirect is a terminal.

Notes

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

```
>>> redirect = io.StringIO()
>>> assert TeeStringIO(redirect).encoding is None
>>> assert TeeStringIO(None).encoding is None
>>> assert TeeStringIO(sys.stdout).encoding is sys.stdout.encoding
>>> redirect = io.TextIOWrapper(io.StringIO())
>>> assert 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.**CaptureStream**
Bases: object

Generic class for capturing streaming output from stdout or stderr

class ubelt.util_stream.**CaptureStdout**(*supress=True*, *enabled=True*)
Bases: *ubelt.util_stream.CaptureStream*

Context manager that captures stdout and stores it in an internal stream

Parameters

- **supress** (*bool, default=True*) – if True, stdout is not printed while captured
- **enabled** (*bool, default=True*) – does nothing if this is False

Example

```
>>> self = CaptureStdout(supress=True)
>>> print('dont capture the table flip (°° ')
>>> with self:
...     text = 'capture the heart '
...     print(text)
>>> print('dont capture look of disapproval _')
>>> import six
>>> assert isinstance(self.text, six.text_type)
>>> assert self.text == text + '\n', 'failed capture text'
```

Example

```
>>> self = CaptureStdout(supress=False)
>>> with self:
...     print('I am captured and printed in stdout')
>>> assert self.text.strip() == 'I am captured and printed in stdout'
```

Example

```
>>> self = CaptureStdout(supress=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

```
>>> CaptureStdout(enabled=False).stop()
>>> CaptureStdout(enabled=True).stop()
```

close()

1.24.1.1.1.24 ubelt.util_time module

This is util_time, it has a few functions for handling time related code that I wish there was standard library support for.

The *timestamp()* is less interesting than the previous two methods, but I have found it useful to have a function that quickly returns an iso8601 timestamp without much fuss.

Timerit is back! But it no longer lives in util_time. Instead it now lives in ubelt/timerit.py

ubelt.util_time.**timestamp**(*method='iso8601'*)
Make an iso8601 timestamp suitable for use in filenames

Parameters **method** (*str, default='iso8601'*) – type of timestamp

Example

```
>>> stamp = timestamp()
>>> print('stamp = {!r}'.format(stamp))
stamp = ...-...-...T...
```

1.24.1.1.2 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.

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*)

Bases: object

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

Writes a file that marks that a procedure has been done by writing a "stamp" file to disk. Removing the stamp file will force recomputation. However, removing or changing the result of the computation may not trigger recomputation unless specific handling is done or the expected "product" of the computation is a file and registered with the stamper. If hasher is None, we only check if the product exists, and we ignore its hash, otherwise it checks that the hash of the product is the same.

Parameters

- **fname** (*str*) – Name of the stamp file
- **cfgstr** (*str*) – Configuration associated with the stamped computation. A common pattern is to call *ubelt.hash_data()* on a dependency list.

  Deprecated in favor of depends. Indicates the state. Either this string or a hash of this string will be used to identify the cache. A cfgstr should always be reasonably readable, thus it is good practice to hash extremely detailed cfgstrs to a reasonable readable level. Use meta to store make original details persist.
- **dpath** (*PathLike*) – Where to store the cached stamp file
- **product** (*PathLike or Sequence[PathLike], optional*) – 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. Defaults to sha1.
- **verbose** (*bool, default=None*) – Passed to internal ub.Cacher object
- **enabled** (*bool, default=True*) – if False, expired always returns True
- **depends** (*str | List[str]*) – 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*) – 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.

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> # Stamp the computation of expensive-to-compute.txt
>>> dpath = ub.ensure_app_cache_dir('ubelt', 'test-cache-stamp')
>>> ub.delete(dpath)
>>> ub.ensuredir(dpath)
>>> product = join(dpath, 'expensive-to-compute.txt')
>>> self = CacheStamp('somedata', cfgstr='someconfig', dpath=dpath,
>>>                   product=product, hasher=None)
>>> self.hasher = None
>>> if self.expired():
>>>     ub.writeto(product, 'very expensive')
>>>     self.renew()
>>> assert not self.expired()
>>> # corrupting the output will not expire in non-robust mode
>>> ub.writeto(product, 'corrupted')
>>> assert not self.expired()
>>> self.hasher = 'sha1'
>>> # but it will expire if we are in robust mode
>>> assert self.expired()
>>> # deleting the product will cause expiration in any mode
>>> self.hasher = None
>>> ub.delete(product)
>>> assert self.expired()
```

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

Check to see if a previously existing stamp is still valid and if the expected result of that computation still exists.

Parameters

- **cfgstr** (*str, optional*) – overrides the instance-level cfgstr
- **product** (*PathLike or Sequence[PathLike], optional*) – override the default product if specified

Returns True if the stamp is invalid or does not exist.

Return type bool

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*)

Bases: object

Cacher designed to be quickly integrated into existing scripts.

A dependency string can be specified, which will invalidate the cache if it changes to an unseen value. The location

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]*) – Indicate dependencies of this cache. If the dependencies change, then the cache is recomputed. New in version 0.8.9, replaces *cfgstr*.
- **dpath** (*PathLike*) – Specifies where to save the cache. If unspecified, Cacher defaults to an application resource dir as given by appname.
- **appname** (*str, default='ubelt'*) – Application name Specifies a folder in the application resource directory where to cache the data if `dpath` is not specified.
- **ext** (*str, default='.pkl'*) – File extension for the cache format
- **meta** (*object*) – Metadata that is also saved with the `cfgstr`. This can be useful to indicate how the `cfgstr` was constructed.
- **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** (*func*) – Overloads the print function. Useful for sending output to loggers (e.g. logging.info, tqdm.tqdm.write, ...)
- **hasher** (*str*) – Type of hashing algorithm to use if `cfgstr` needs to be condensed to less than 49 characters.
- **protocol** (*int, default=2*) – Protocol version used by pickle. If python 2 compatibility is not required, then it is better to use protocol 4.
- **cfgstr** (*str*) – Deprecated in favor of depends. Indicates the state. Either this string or a hash of this string will be used to identify the cache. A cfgstr should always be reasonably readable, thus it is good practice to hash extremely detailed cfgstrs to a reasonable readable level. Use meta to store make original details persist.

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()
>>> 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

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, optional*) – overrides the instance-level cfgstr

Returns 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', cfgstr='cfg1')
>>> self.get_fpath()
>>> self = Cacher('test_cacher3', cfgstr='cfg1' * 32)
>>> self.get_fpath()
```

exists(*cfgstr=None*)

Check to see if the cache exists

Parameters **cfgstr** (*str, optional*) – 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()
>>> cacher = Cacher('versioned_data_v2', depends='1')
>>> cacher.ensure(lambda: 'data1')
>>> known_fpaths.add(cacher.get_fpath())
>>> cacher = Cacher('versioned_data_v2', depends='2')
>>> 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')
>>> 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, optional*) – 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, optional*) – 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, optional*) – 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(cfgstr)*.

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, optional*) – overrides the instance-level cfgstr

Example

```
>>> from ubelt.util_cache import *  # NOQA
>>> # Normal functioning
>>> cfgstr = 'long-cfg' * 32
>>> cacher = Cacher('test_enabled_save', cfgstr)
>>> cacher.save('data')
>>> assert exists(cacher.get_fpath()), 'should be enabeled'
>>> 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'
>>> cfgstr = 'func params'
>>> cacher = Cacher(fname, cfgstr)
>>> cacher.clear()
>>> data1 = cacher.ensure(func)
>>> data2 = cacher.ensure(func)
>>> assert data1 == 'expensive result'
>>> assert data1 == data2
>>> cacher.clear()
```

class ubelt.**CaptureStdout** (*supress=True*, *enabled=True*)
Bases: *ubelt.util_stream.CaptureStream*

Context manager that captures stdout and stores it in an internal stream

Parameters

- **supress** (*bool, default=True*) – if True, stdout is not printed while captured
- **enabled** (*bool, default=True*) – does nothing if this is False

Example

```
>>> self = CaptureStdout(supress=True)
>>> print('dont capture the table flip (°° ')
>>> with self:
...     text = 'capture the heart '
...     print(text)
>>> print('dont capture look of disapproval _')
>>> import six
>>> assert isinstance(self.text, six.text_type)
>>> assert self.text == text + '\n', 'failed capture text'
```

Example

```
>>> self = CaptureStdout(supress=False)
>>> with self:
...     print('I am captured and printed in stdout')
>>> assert self.text.strip() == 'I am captured and printed in stdout'
```

Example

```
>>> self = CaptureStdout(supress=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

```
>>> CaptureStdout(enabled=False).stop()
>>> CaptureStdout(enabled=True).stop()
```

close()

class ubelt.**CaptureStream**
Bases: object

Generic class for capturing streaming output from stdout or stderr

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 explicilty 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.**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.**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*, ***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.
- **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()
```

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.

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 progres 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

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

```
>>> redirect = io.StringIO()
>>> self = TeeStringIO(redirect)
```

isatty()
Returns true of the redirect is a terminal.

Notes

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

```
>>> redirect = io.StringIO()
>>> assert TeeStringIO(redirect).encoding is None
>>> assert TeeStringIO(None).encoding is None
>>> assert TeeStringIO(sys.stdout).encoding is sys.stdout.encoding
>>> redirect = io.TextIOWrapper(io.StringIO())
>>> assert 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.

Example

```
>>> with TempDir() as self:
>>>     dpath = self.dpath
>>>     assert exists(dpath)
>>> assert not exists(dpath)
```

Example

```
>>> 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

class ubelt.**Timerit**(*num=1*, *label=None*, *bestof=3*, *unit=None*, *verbose=None*)
Bases: object

Reports the average time to run a block of code.

Unlike *%timeit*, *Timerit* can handle multiline blocks of code. It runs inline, and doesn't depend on magic or strings. Just indent your code and place in a Timerit block. See https://github.com/Erotemic/vimtk for vim functions that will insert one of these in for you (ok that part is a little magic).

Parameters

- **num** (*int, default=1*) – number of times to run the loop

- **label** (*str, default=None*) – identifier for printing
- **bestof** (*int, default=3*) – takes the max over this number of trials
- **unit** (*str*) – what units time is reported in
- **verbose** (*int*) – verbosity flag, defaults to True if label is given

Variables

- **- labeled measurements taken by this object** (*measures*) –
- **- ranked measurements** (*rankings*) –

Example

```
>>> import math
>>> num = 3
>>> t1 = Timerit(num, label='factorial', verbose=1)
>>> for timer in t1:
>>>     # <write untimed setup code here> this example has no setup
>>>     with timer:
>>>         # <write code to time here> for example...
>>>         math.factorial(100)
Timed best=..., mean=... for factorial
>>> # <you can now access Timerit attributes>
>>> assert t1.total_time > 0
>>> assert t1.n_loops == t1.num
>>> assert t1.n_loops == num
```

Example

```
>>> # xdoc: +IGNORE_WANT
>>> import math
>>> num = 4
>>> # If the timer object is unused, time will still be recorded,
>>> # but with less precision.
>>> for _ in Timerit(num, 'concise', bestof=2, verbose=2):
>>>     math.factorial(100)
Timed concise for: 4 loops, best of 2
    time per loop: best=1.637 µs, mean=1.935 ± 0.3 µs
>>> # Using the timer object results in the most precise timings
>>> for timer in Timerit(num, 'precise', bestof=2, verbose=3):
>>>     with timer: math.factorial(100)
Timed precise for: 4 loops, best of 2
    body took: 8.696 µs
    time per loop: best=1.754 µs, mean=1.821 ± 0.1 µs
```

reset (*label=None*, *measures=False*)

clears all measurements, allowing the object to be reused

Parameters

- **label** (*str, optional*) – change the label if specified
- **measures** (*bool, default=False*) – if True reset measures

Example

```
>>> import math
>>> ti = Timerit(num=10, unit='us', verbose=True)
>>> _ = ti.reset(label='10!').call(math.factorial, 10)
Timed best=...s, mean=...s for 10!
>>> _ = ti.reset(label='20!').call(math.factorial, 20)
Timed best=...s, mean=...s for 20!
>>> _ = ti.reset().call(math.factorial, 20)
Timed best=...s, mean=...s for 20!
>>> _ = ti.reset(measures=True).call(math.factorial, 20)
```

call (*func*, **args*, ***kwargs*)

Alternative way to time a simple function call using condensed syntax.

Returns

Use *min*, or *mean* to get a scalar. Use *print* to output a report to stdout.

Return type self (*Timerit*)

Example

```
>>> import math
>>> time = Timerit(num=10).call(math.factorial, 50).min()
>>> assert time > 0
```

property rankings

Orders each list of measurements by ascending time

Example

```
>>> import math
>>> ti = Timerit(num=1)
>>> _ = ti.reset('a').call(math.factorial, 5)
>>> _ = ti.reset('b').call(math.factorial, 10)
>>> _ = ti.reset('c').call(math.factorial, 20)
>>> ti.rankings
>>> ti.consistency
```

property consistency

” Take the hamming distance between the preference profiles to as a measure of consistency.

min ()

The best time overall.

This is typically the best metric to consider when evaluating the execution time of a function. To understand why consider this quote from the docs of the original timeit module:

‘” In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python’s speed, but by other processes interfering with your timing accuracy. So the min() of the result is probably the only number you should be interested in. ‘”

Returns minimum measured seconds over all trials

Return type float

Example

```
>>> import math
>>> self = Timerit(num=10, verbose=0)
>>> self.call(math.factorial, 50)
>>> assert self.min() > 0
```

mean()

The mean of the best results of each trial.

Returns mean of measured seconds

Return type float

Note: This is typically less informative than simply looking at the min. It is recommended to use min as the expectation value rather than mean in most cases.

Example

```
>>> import math
>>> self = Timerit(num=10, verbose=0)
>>> self.call(math.factorial, 50)
>>> assert self.mean() > 0
```

std()

The standard deviation of the best results of each trial.

Returns standard deviation of measured seconds

Return type float

Note: As mentioned in the timeit source code, the standard deviation is not often useful. Typically the minimum value is most informative.

Example

```
>>> import math
>>> self = Timerit(num=10, verbose=1)
>>> self.call(math.factorial, 50)
>>> assert self.std() >= 0
```

report(*verbose=1*)

Creates a human readable report

Parameters **verbose** (*int*) – verbosity level. Either 1, 2, or 3.

Returns the report

Return type str

SeeAlso: *Timerit.print()*

Example

```
>>> import math
>>> ti = Timerit(num=1).call(math.factorial, 5)
>>> print(ti.report(verbose=1))
Timed best=...s, mean=...s
```

print(*verbose=1*)

Prints human readable report using the print function

Parameters **verbose** (*int*) – verbosity level

SeeAlso: *Timerit.report()*

Example

```
>>> import math
>>> Timerit(num=10).call(math.factorial, 50).print(verbose=1)
Timed best=...s, mean=...s
>>> Timerit(num=10).call(math.factorial, 50).print(verbose=2)
Timed for: 10 loops, best of 3
    time per loop: best=...s, mean=...s
>>> Timerit(num=10).call(math.factorial, 50).print(verbose=3)
Timed for: 10 loops, best of 3
    body took: ...
    time per loop: best=...s, mean=...s
```

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

Determine if all items in a sequence are the same

Parameters

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

Returns True if all items are equal, otherwise False

Return type bool

Example

```
>>> allsame([1, 1, 1, 1])
True
>>> allsame([])
True
>>> allsame([0, 1])
False
>>> iterable = iter([0, 1, 1, 1])
>>> next(iterable)
>>> allsame(iterable)
True
>>> allsame(range(10))
False
>>> 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[B] | Mapping[A, B]*) – indexable to sort by
- **key** (*Callable[[A], B], default=None*) – customizes the ordering of the indexable

Returns the index of the item with the maximum value.

Return type int

Example

```
>>> assert argmax({'a': 3, 'b': 2, 'c': 100}) == 'c'
>>> assert argmax(['a', 'c', 'b', 'z', 'f']) == 3
>>> assert argmax([[0, 1], [2, 3, 4], [5]], key=len) == 1
>>> assert argmax({'a': 3, 'b': 2, 3: 100, 4: 4}) == 3
>>> assert 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[B] | Mapping[A, B]*) – indexable to sort by
- **key** (*Callable[[A], B], default=None*) – customizes the ordering of the indexable

Returns the index of the item with the minimum value.

Return type int

Example

```
>>> assert argmin({'a': 3, 'b': 2, 'c': 100}) == 'b'
>>> assert argmin(['a', 'c', 'b', 'z', 'f']) == 0
>>> assert argmin([[0, 1], [2, 3, 4], [5]], key=len) == 2
>>> assert argmin({'a': 3, 'b': 2, 3: 100, 4: 4}) == 'b'
>>> assert 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[B] | Mapping[A, B]*) – indexable to sort by
- **key** (*Callable[[A], B], default=None*) – customizes the ordering of the indexable
- **reverse** (*bool, default=False*) – if True returns in descending order

Returns indices - list of indices such that sorts the indexable

Return type List[int]

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[V]*) – indexable collection of items
- **key** (*Callable[[V], 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

```
>>> items = [0, 2, 5, 1, 1, 0, 2, 4]
>>> indices = list(argunique(items))
>>> assert indices == [0, 1, 2, 3, 7]
>>> indices = list(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** (*object, default=NoParam*) – a value to return if not specified.
- **argv** (*Optional[list], 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

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*, *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, default=''*) – placed between the basename and extension
- **prefix** (*str, default=''*) – placed in front of the basename
- **ext** (*str, default=None*) – if specified, replaces the extension
- **base** (*str, default=None*) – if specified, replaces the basename without extension
- **dpath** (*str | PathLike, default=None*) – if specified, replaces the specified "relative" directory, which by default is the parent directory.
- **relative** (*str | PathLike, default=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, default=False*) – 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

```
>>> 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
```

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'*)

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

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

References

http://stackoverflow.com/questions/434287/iterate-over-a-list-in-chunks

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))) == 2
>>> assert len(list(ub.chunks([], 2, None, 'none'))) == 0
>>> assert len(list(ub.chunks([], 2, None, 'cycle'))) == 0
>>> assert len(list(ub.chunks([], 2, None, 'replicate'))) == 0
```

Example

```
>>> 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

```
>>> 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))
```

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*, ***kwargs*)

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 or Sequence*) – 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, optional*) – 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** (*PathLike, optional*) – path to run command
- **env** (*str, optional*) – environment passed to Popen
- **tee_backend** (*str, optional*) – 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.
- ****kwargs** – only used to support deprecated arguments

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

Notes

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 [3]_ for a potential cross-platform shlex solution for windows.

CommandLine: python -m ubelt.util_cmd cmd python -c "import ubelt as ub; ub.cmd('ping localhost -c 2', verbose=2)"

References

Example

```
>>> info = 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

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

Example

```
>>> # windows echo will output extra single quotes
>>> info = 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
>>> info = cmd('echo str&&echo shell', verbose=0, shell=True)
>>> assert info['out'].strip() == 'str' + chr(10) + 'shell'
```

Example

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

Example

```
>>> import pytest
>>> info = cmd('echo hi', check=True)
>>> import subprocess
>>> with pytest.raises(subprocess.CalledProcessError):
>>>     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)
>>> info1 = ub.cmd(('touch', fpath1), detach=True)
>>> info2 = ub.cmd('echo writing2 > ' + fpath2, shell=True, detach=True)
>>> while not exists(fpath1):
...     pass
>>> while not exists(fpath2):
...     pass
>>> assert ub.readfrom(fpath1) == ''
>>> assert ub.readfrom(fpath2).strip() == 'writing2'
>>> info1['proc'].wait()
>>> info2['proc'].wait()
```

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

```
>>> from ubelt.util_str import *  # NOQA
>>> # Simulate an indented part of code
>>> if True:
>>>     # notice the indentation on this will be normal
>>>     codeblock_version = codeblock(
...             '''
...             def foo():
...                 return 'bar'
...             '''
...         )
>>>     # notice the indentation and newlines on this will be odd
>>>     normal_version = ('''
...         def foo():
...             return 'bar'
```

(continues on next page)

(continued from previous page)

```
...     ''')
>>> 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.**compress**(*items*, *flags*)

Selects from `items` where the corresponding value in `flags` is True. This is similar to `numpy.compress()`.

This is actually a simple alias for `itertools.compress()`.

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]

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.**compressuser**(*path*, *home='~'*)
Inverse of `os.path.expanduser()`.

Parameters

- **path** (*str | PathLike*) – path in system file structure
- **home** (*str, default='~'*) – 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 tilde

Return type str

Example

```
>>> from os.path import join
>>> path = expanduser('~')
>>> assert path != '~'
>>> assert compressuser(path) == '~'
>>> assert compressuser(path + '1') == path + '1'
>>> assert compressuser(path + '/1') == join('~', '1')
>>> assert compressuser(path + '/1', '$HOME') == join('$HOME', '1')
```

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. https://github.com/hsoft/send2trash

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)
```

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

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

Parameters ***args** – a sequence of dictionaries (or sets of keys)

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

Return type Dict | OrderedDict

Todo:

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

Example

```
>>> dict_diff({'a': 1, 'b': 1}, {'a'}, {'c'})
{'b': 1}
>>> dict_diff(odict([('a', 1), ('b', 2)]), odict([('c', 3)]))
OrderedDict([('a', 1), ('b', 2)])
>>> dict_diff()
{}
>>> 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*)

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

Parameters ***args** – a sequence of dictionaries (or sets of keys)

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

Return type Dict | OrderedDict

Notes

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

```
>>> dict_isect({'a': 1, 'b': 2, 'c': 3}, ['a', 'c', 'd'])
{'a': 1, 'c': 3}
```

Example

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

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

Get a subset of a dictionary

Parameters

- **dict_** (*Dict[A, B]*) – superset dictionary
- **keys** (*Iterable[A]*) – keys to take from `dict_`
- **default** (*object, optional*) – if specified uses default if keys are missing
- **cls** (*type, default=OrderedDict*) – type of the returned dictionary.

Returns subset dictionary

Return type cls[A, B]

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_take**(*dict_*, *keys*, *default=NoParam*)

Generates values from a dictionary

Parameters

- **dict_** (*Mapping*) – a dictionary to take from
- **keys** (*Iterable*) – the keys to take
- **default** (*object, optional*) – if specified uses default if keys are missing

Example

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

ubelt.**dict_union**(**args*)

Combines the disjoint keys in multiple dictionaries. For intersecting keys, dictionaries towards the end of the sequence are given precedence.

Parameters ***args** – a sequence of dictionaries

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

Return type Dict | OrderedDict

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

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

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

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** (*PathLike | io.BytesIOtringIO*) – 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** (*PathLike*) – where to download the file. If unspecified *appname* is used to determine this. Mutually exclusive with fpath.
- **fname** (*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.

Returns fpath - path to the downloaded file.

Return type PathLike

Raises

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

Notes

Based largely on code in pytorch[4] with modifications influenced by other resources [1]_ [2]_ [3]_.

References

Todo:

- [ ] fine-grained control of progress

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
```

[4] https://github.com/pytorch/pytorch/blob/2787f1d8edbd4aadd4a8680d204341a1d7112e2d/torch/hub.py#L347

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> import ubelt as ub
>>> import io
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> file = io.BytesIO()
>>> fpath = 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)
>>> # test download from girder
>>> import pytest
>>> import ubelt as ub
>>> 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[A]*) – full sequence
- **items2** (*Iterable[B]*) – 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[A, B]

Example

```
>>> assert dzip([1, 2, 3], [4]) == {1: 4, 2: 4, 3: 4}
>>> assert dzip([1, 2, 3], [4, 4, 4]) == {1: 4, 2: 4, 3: 4}
>>> assert dzip([], [4]) == {}
```

ubelt.**editfile**(*fpath*, *verbose=True*)

DEPRICATED: This has been ported to xdev, please use that version.

Opens a file or code corresponding to a live python object in your preferred visual editor. This function is mainly useful in an interactive IPython session.

The visual editor is determined by the *VISUAL* environment variable. If this is not specified it defaults to gvim.

Parameters

- **fpath** (*PathLike*) – a file path or python module / function
- **verbose** (*int*) – verbosity

Example

```
>>> # xdoctest: +SKIP
>>> # This test interacts with a GUI frontend, not sure how to test.
>>> import ubelt as ub
>>> ub.editfile(ub.util_platform.__file__)
>>> ub.editfile(ub)
>>> ub.editfile(ub.editfile)
```

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_app_resource_dir**(*appname*, **args*)
Calls *get_app_resource_dir* but ensures the directory exists.

DEPRICATED in favor of ensure_app_config_dir / ensure_app_data_dir

Parameters

- **appname** (*str*) – the name of the application
- ***args** – any other subdirectories may be specified

SeeAlso: get_app_resource_dir

ubelt.**ensure_unicode**(*text*)
Casts bytes into utf8 (mostly for python2 compatibility)

References

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=None*, *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, default=0o1777*) – octal mode of directory
- **verbose** (*int, default=0*) – verbosity
- **recreate** (*bool, default=False*) – 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

Notes

This function is not thread-safe in Python2

Example

```
>>> from ubelt.util_platform 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.

Less aggressive than truepath. Only expands environs and tilde. Does not change relative paths to absolute paths.

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

Returns expanded path

Return type str

Example

```
>>> 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

List[int]]: maps each duplicate item to the indices at which it appears

Return type dict[T

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], 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+.

Notes

This is essentially the `which` UNIX command

References

https://stackoverflow.com/questions/377017/test-if-executable-exists-in-python/377028#377028 https://docs.python.org/dev/library/shutil.html#shutil.which

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

```
>>> assert not find_exe('noexist', multi=False)
>>> assert find_exe('ping', multi=False)
>>> assert not find_exe('noexist', multi=True)
>>> assert find_exe('ping', 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

- **fname** (*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`

Notes

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

Notes

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('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.

This is simply an alias for `itertools.chain.from_iterable()`.

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

Returns flattened items

Return type Iterable[Any]

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.**get_app_resource_dir**(*appname*, **args*)
Returns a writable directory for an application This should be used for persistent configuration files.

DEPRICATED in favor of get_app_config_dir / get_app_data_dir

Parameters

- **appname** (*str*) – the name of the application
- ***args** – any other subdirectories may be specified

Returns dpath: writable resource directory for this application

Return type PathLike

SeeAlso: ensure_app_resource_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 to the file to download
- **fpath** (*PathLike*) – The full path to download the file to. If unspecified, the arguments *dpath* and *fname* are used to determine this.
- **dpath** (*PathLike*) – where to download the file. If unspecified *appname* is used to determine this. Mutually exclusive with fpath.
- **fname** (*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.

- ****download_kw** – additional kwargs to pass to ub.download

Returns fpath - path to downloaded or cached file.

Return type 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
>>> fname = 'foo.bar'
>>> url = 'http://i.imgur.com/rqwaDag.png'
>>> prefix1 = '944389a39dfb8fa9'
>>> fpath = ub.grabdata(url, fname=fname, hash_prefix=prefix1)
>>> stamp_fpath = fpath + '.sha512.hash'
>>> assert ub.readfrom(stamp_fpath) == 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
>>> open(stamp_fpath, 'w').write('corrupt-stamp')
>>> fpath = ub.grabdata(url, fname=fname, hash_prefix=prefix1)
>>> assert ub.readfrom(stamp_fpath) == prefix1
>>> #
>>> # Check that a redownload occurs when the stamp is removed
>>> ub.delete(stamp_fpath)
>>> open(fpath, 'w').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'
>>> fpath = ub.grabdata(url2, fname=fname, hash_prefix=prefix2)
>>> assert ub.readfrom(stamp_fpath) == prefix2
```

ubelt.**group_items**(*items*, *key*)
Groups a list of items by group id.

Parameters

- **items** (*Iterable[A]*) – a list of items to group
- **key** (*Iterable[B] | Callable[[A], B]*) – 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[B, List[A]]

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*, *hashlen=NoParam*, *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 | hashlib.HASH, 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.
- **hashlen** (*int*) – Maximum number of symbols in the returned hash. If not specified, all are returned. DEPRECATED. Use slice syntax instead.
- **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.

Notes

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

Notes

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*, *hashlen=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 dependant on blocksize.
- **maxbytes** (*int | None*) – if specified, only hash the leading *maxbytes* of data in the file.
- **hasher** (*str | hashlib.HASH, 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
- **hashlen** (*int*) – maximum number of symbols in the returned hash. If not specified, all are returned. DEPRECATED. DO NOT USE.
- **base** (*List[str] | str, default='hex'*) – list of symbols or shorthand key. Valid keys are 'abc', 'hex', and 'dec'.

Notes

For better hashes keep stride = 1 For faster hashes set stride > 1 blocksize matters when stride > 1

References

http://stackoverflow.com/questions/3431825/md5-checksum-of-a-file http://stackoverflow.com/questions/5001893/when-to-use-sha-1-vs-sha-2

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> fpath = join(ub.ensure_app_cache_dir('ubelt'), 'tmp.txt')
>>> ub.writeto(fpath, 'foobar')
>>> print(ub.hash_file(fpath, hasher='sha1', base='hex'))
8843d7f92416211de9ebb963ff4ce28125932878
```

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> fpath = join(ub.ensure_app_cache_dir('ubelt'), 'tmp.txt')
>>> ub.writeto(fpath, 'foobar')
>>> print(ub.hash_file(fpath, hasher='sha1', base='hex', maxbytes=1000))
8843d7f92416211de9ebb963ff4ce28125932878
```

```
>>> # We have the ability to only hash at most ``maxbytes`` in a file
>>> ub.writeto(fpath, '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 dependant 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)
>>> assert h1 != h2 != h3
>>> assert h1 == h0
>>> assert h2 == h4
```

Example

```
>>> import ubelt as ub
>>> from os.path import join
>>> fpath = ub.touch(join(ub.ensure_app_cache_dir('ubelt'), 'empty_file'))
>>> # Test that the output is the same as sha1sum
>>> 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:

```
>>> from ubelt.util_str import *
>>> 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 (__name__)

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

Returns module

Return type module

Example

```
>>> # test with modules that wont be imported in normal circumstances
>>> # todo write a test where we gaurentee 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** (*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 module

References

https://stackoverflow.com/questions/67631/import-module-given-path

Notes

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.

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
>>> 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 module.__name__ == os.path.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

```
>>> from ubelt.util_str import *  # NOQA
>>> NL = chr(10)  # newline character
>>> text = 'Lorem ipsum' + NL + 'dolor sit amet'
>>> prefix = '    '
>>> result = indent(text, prefix)
>>> assert all(t.startswith(prefix) for t in result.split(NL))
```

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[[T, ...], 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

```
>>> class Foo(object):
>>>     def bar(self):
>>>         return 'bar'
>>> def baz(self):
>>>     return 'baz'
>>> self = Foo()
>>> assert self.bar() == 'bar'
>>> assert not hasattr(self, 'baz')
>>> inject_method(self, baz)
>>> assert not hasattr(Foo, 'baz'), 'should only change one instance'
>>> assert self.baz() == 'baz'
>>> 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[A, B]*) – 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[B, A] | Dict[B, Set[A]]

Notes

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 overlaping windows in a sequence

Return type Iterable[T]

Example

```
>>> iterable = [1, 2, 3, 4, 5, 6]
>>> size, step, wrap = 3, 1, True
>>> window_iter = 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

```
>>> iterable = [1, 2, 3, 4, 5, 6]
>>> size, step, wrap = 3, 2, True
>>> window_iter = 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

```
>>> iterable = [1, 2, 3, 4, 5, 6]
>>> size, step, wrap = 3, 2, False
>>> window_iter = 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

```
>>> iterable = []
>>> size, step, wrap = 3, 2, False
>>> window_iter = 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

```
>>> obj_list = [3, [3], '3', (3,), [3, 4, 5], {}]
>>> result = [iterable(obj) for obj in obj_list]
>>> assert result == [False, True, False, True, True, True]
>>> result = [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[[A], C] | Mapping[A, C]*) – a function or indexable object
- **dict_** (*Dict[A, B]*) – a dictionary

Returns transformed dictionary

Return type Dict[C, B]

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

Example

```
>>> dict_ = {'a': [1, 2, 3], 'b': []}
>>> func = ord
>>> newdict = map_keys(func, dict_)
>>> print(newdict)
>>> assert newdict == {97: [1, 2, 3], 98: []}
>>> dict_ = {0: [1, 2, 3], 1: []}
>>> func = ['a', 'b']
>>> newdict = 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[[B], C] | Mapping[B, C]*) – a function or indexable object
- **dict_** (*Dict[A, B]*) – a dictionary

Returns transformed dictionary

Return type Dict[A, C]

Example

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

Example

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

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

memoization decorator that respects args and kwargs

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

Returns memoized wrapper

Return type Callable

References

https://wiki.python.org/moin/PythonDecoratorLibrary#Memoize

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.**memoize_method**(*func*)

Bases: object

memoization decorator for a method that respects args and kwargs

References

http://code.activestate.com/recipes/577452-a-memoize-decorator-for-instance-methods/

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.**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.

Notes

implementation is a modified version of [1].

References

..[1] https://github.com/estebistec/python-memoized-property

Example

```
>>> class C(object):
...     load_name_count = 0
...     @memoize_property
...     def name(self):
...         "name's docstring"
...         self.load_name_count += 1
...         return "the name"
...     @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.**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

```
>>> from ubelt.util_str import *  # NOQA
>>> text = (
>>>     '''
>>>     Lorem ipsum dolor sit amet, consectetur adipiscing
>>>     elit, sed do eiusmod tempor incididunt ut labore et
>>>     dolore magna aliqua.
>>>     ''')
>>> out = 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*)
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** (*List[T]*) – an iterable

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

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
```

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 cahce 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.**platform_resource_dir**()
Alias for platform_cache_dir

DEPRICATED in favor of platform_config_dir / platform_data_dir

Returns a directory which should be writable for any application This should be used for persistent configuration files.

Returns path to the resource dir used by the current operating system

Return type PathLike

ubelt.**readfrom**(*fpath*, *aslines=False*, *errors='replace'*, *verbose=None*)
Reads (utf8) text from a file.

Note: You probably should use *open(<fpath>).read()* 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 very configurable. By default it aims to produce strings that are executable and consistent between Python versions. This makes them great for doctests.

Notes

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

Parameters

- **data** (*object*) – an arbitrary python object
- ****kwargs** – see "the Kwargs" section

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, ...)`.

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 determenistic 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

Notes

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

Example

```
>>> from ubelt.util_format import *
>>> 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')]),
... }
```

(continues on next page)

(continued from previous page)

```
>>> # In the interest of saving space we are only going to show the
>>> # output for the first example.
>>> result = 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}],
    '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 = repr2(dict_, nl=3, precision=2); print(result)
>>> result = repr2(dict_, nl=2, precision=2); print(result)
>>> result = repr2(dict_, nl=1, precision=2, itemsep='', explicit=True);␣
↪print(result)
>>> result = repr2(dict_, nl=1, precision=2, nobr=1, itemsep='', explicit=True);␣
↪print(result)
>>> result = repr2(dict_, nl=3, precision=2, cbr=True); print(result)
>>> result = repr2(dict_, nl=3, precision=2, si=True); print(result)
>>> result = repr2(dict_, nl=3, sort=True); print(result)
>>> result = repr2(dict_, nl=3, sort=False, trailing_sep=False); print(result)
>>> result = repr2(dict_, nl=3, sort=False, trailing_sep=False, nobr=True);␣
↪print(result)
```

Example

```
>>> from ubelt.util_format import *
>>> 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 = repr2(dict_, nl=6, precision=2, cbr=1)
>>> print('---')
>>> print(result)
>>> result = repr2(dict_, nl=-1, precision=2)
>>> print('---')
>>> print(result)
```

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

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

Parameters

- **path** (*str | PathLike*) – path in system file structure
- **home** (*str, default='~'*) – 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 tilde

Return type str

Example

```
>>> 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[A, B]*) – dictionary to sort. The keys must be of comparable types.
- **key** (*Callable[[A], Any], optional*) – 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[A, B]

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

Return an ordered dictionary sorted by its values

Parameters

- **dict_** (*Dict[A, B]*) – dictionary to sort. The values must be of comparable types.
- **key** (*Callable[[B], Any], optional*) – 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[A, B]

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_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

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.**startfile**(*fpath*, *verbose=True*)

Uses default program defined by the system to open a file. This is done via *os.startfile* on windows, *open* on mac, and *xdg-open* on linux.

Parameters

- **fpath** (*str | PathLike*) – a file to open using the program associated with the files extension type.
- **verbose** (*int*) – verbosity

References

http://stackoverflow.com/questions/2692873/quote-posix

Example

```
>>> # xdoctest: +SKIP
>>> # This test interacts with a GUI frontend, not sure how to test.
>>> import ubelt as ub
>>> base = ub.ensure_app_cache_dir('ubelt')
>>> fpath1 = join(base, 'test_open.txt')
>>> ub.touch(fpath1)
>>> proc = ub.startfile(fpath1)
```

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

Create a symbolic link.

This will work on linux or windows, however windows does have some corner cases. For more details see notes in `ubelt._win32_links`.

Parameters

- **path** (*PathLike*) – path to real file or directory
- **link_path** (*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 PathLike

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*)

Selects a subset of a list based on a list of indices.

This is similar to `np.take`, but pure python. This also supports specifying a default element if `items` is an iterable of dictionaries.

Parameters

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

Yeilds: V: a selected item within the list

SeeAlso: `ub.dict_subset()`

Notes

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

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.**timestamp**(*method='iso8601'*)

Make an iso8601 timestamp suitable for use in filenames

Parameters **method** (*str, default='iso8601'*) – type of timestamp

Example

```
>>> stamp = timestamp()
>>> print('stamp = {!r}'.format(stamp))
stamp = ...-...-...T...
```

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** (*file*) – 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

https://stackoverflow.com/questions/1158076/implement-touch-using-python

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.`**`truepath`**(*path*, *real=False*)

Normalizes a string representation of a path and does shell-like expansion.

Parameters

- **path** (*str | PathLike*) – string representation of a path
- **real** (*bool*) – if True, all symbolic links are followed. (default: False)

Returns normalized path

Return type str

Note: This function is similar to the composition of expanduser, expandvars, normpath, and (realpath if *real* else abspath). However, on windows backslashes are then replaced with forward slashes to offer a consistent unix-like experience across platforms.

On windows expanduser will expand environment variables formatted as %name%, whereas on unix, this will not occur.

Example

```
>>> from os.path import join
>>> import ubelt as ub
>>> assert ub.truepath('~/foo') == join(ub.userhome(), 'foo')
>>> assert ub.truepath('~/foo') == ub.truepath('~/foo/bar/..')
>>> assert ub.truepath('~/foo', real=True) == ub.truepath('~/foo')
```

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

Generates unique items in the order they appear.

Parameters

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

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

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
>>> import six
>>> items = ['A', 'a', 'b', 'B', 'C', 'c', 'D', 'e', 'D', 'E']
>>> unique_items = list(ub.unique(items, key=six.text_type.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*) – indexable collection of items
- **key** (*Callable[[V], Any], 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 = unique_flags(items)
>>> assert flags == [True, True, True, False, False, True, False]
>>> flags = 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, default=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

```
>>> import getpass
>>> username = getpass.getuser()
>>> assert userhome() == expanduser('~')
>>> assert userhome(username) == expanduser('~')
```

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: You probably should 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: It turns out that *open(<fpath>).write(<to_write>)* does not work in pypy. See *https://pypy.org/compat.html*. This is a strong argument for keeping this function.

Example

```
>>> import ubelt as ub
>>> dpath = ub.ensure_app_cache_dir('ubelt')
>>> fpath = dpath + '/' + 'testwrite.txt'
>>> if exists(fpath):
>>>     os.remove(fpath)
>>> to_write = 'utf-8 symbols , , , , , , , , and .'
>>> 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
>>> 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']
>>> 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
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

u

ubelt, 97
ubelt.__init__, 1
ubelt.orderedset, 8
ubelt.progiter, 13
ubelt.timerit, 19
ubelt.util_arg, 24
ubelt.util_cache, 26
ubelt.util_cmd, 33
ubelt.util_colors, 35
ubelt.util_const, 37
ubelt.util_dict, 37
ubelt.util_download, 46
ubelt.util_format, 50
ubelt.util_func, 54
ubelt.util_hash, 55
ubelt.util_import, 59
ubelt.util_io, 64
ubelt.util_links, 66
ubelt.util_list, 68
ubelt.util_memoize, 77
ubelt.util_mixins, 81
ubelt.util_path, 83
ubelt.util_platform, 87
ubelt.util_str, 91
ubelt.util_stream, 94
ubelt.util_time, 96

INDEX

A

add() (*ubelt.OrderedSet method*), 107
add() (*ubelt.orderedset.OrderedSet method*), 9
allsame() (*in module ubelt*), 121
allsame() (*in module ubelt.util_list*), 75
append() (*ubelt.OrderedSet method*), 107
append() (*ubelt.orderedset.OrderedSet method*), 9
argflag() (*in module ubelt*), 121
argflag() (*in module ubelt.util_arg*), 25
argmax() (*in module ubelt*), 122
argmax() (*in module ubelt.util_list*), 76
argmin() (*in module ubelt*), 122
argmin() (*in module ubelt.util_list*), 76
argsort() (*in module ubelt*), 123
argsort() (*in module ubelt.util_list*), 75
argunique() (*in module ubelt*), 123
argunique() (*in module ubelt.util_list*), 72
argval() (*in module ubelt*), 124
argval() (*in module ubelt.util_arg*), 24
augpath() (*in module ubelt*), 125
augpath() (*in module ubelt.util_path*), 83
AutoDict (*class in ubelt*), 97
AutoDict (*class in ubelt.util_dict*), 38
AutoOrderedDict (*class in ubelt*), 97
AutoOrderedDict (*class in ubelt.util_dict*), 38

B

begin() (*ubelt.ProgIter method*), 114
begin() (*ubelt.progiter.ProgIter method*), 17
boolmask() (*in module ubelt*), 126
boolmask() (*in module ubelt.util_list*), 73

C

Cacher (*class in ubelt*), 99
Cacher (*class in ubelt.util_cache*), 27
CacheStamp (*class in ubelt*), 98
CacheStamp (*class in ubelt.util_cache*), 31
call() (*ubelt.Timerit method*), 119
call() (*ubelt.timerit.Timerit method*), 22
CaptureStdout (*class in ubelt*), 103
CaptureStdout (*class in ubelt.util_stream*), 95
CaptureStream (*class in ubelt*), 104
CaptureStream (*class in ubelt.util_stream*), 95
chunks (*class in ubelt*), 126
chunks (*class in ubelt.util_list*), 68
cleanup() (*ubelt.TempDir method*), 116
cleanup() (*ubelt.util_path.TempDir method*), 83
clear() (*ubelt.Cacher method*), 102
clear() (*ubelt.OrderedSet method*), 109
clear() (*ubelt.orderedset.OrderedSet method*), 11
clear() (*ubelt.util_cache.Cacher method*), 29
close() (*ubelt.CaptureStdout method*), 104
close() (*ubelt.util_stream.CaptureStdout method*), 96
cmd() (*in module ubelt*), 128
cmd() (*in module ubelt.util_cmd*), 33
codeblock() (*in module ubelt*), 130
codeblock() (*in module ubelt.util_str*), 92
color_text() (*in module ubelt*), 131
color_text() (*in module ubelt.util_colors*), 36
compress() (*in module ubelt*), 131
compress() (*in module ubelt.util_list*), 71
compressuser() (*in module ubelt*), 132
consistency() (*ubelt.Timerit property*), 119
consistency() (*ubelt.timerit.Timerit property*), 22
copy() (*ubelt.OrderedSet method*), 107
copy() (*ubelt.orderedset.OrderedSet method*), 9
cycle() (*ubelt.chunks static method*), 128
cycle() (*ubelt.util_list.chunks static method*), 70

D

ddict (*in module ubelt*), 132
ddict (*in module ubelt.util_dict*), 39
delete() (*in module ubelt*), 132
delete() (*in module ubelt.util_io*), 65
dict_diff() (*in module ubelt*), 133
dict_diff() (*in module ubelt.util_dict*), 41
dict_hist() (*in module ubelt*), 133
dict_hist() (*in module ubelt.util_dict*), 39
dict_isect() (*in module ubelt*), 134
dict_isect() (*in module ubelt.util_dict*), 41
dict_subset() (*in module ubelt*), 135
dict_subset() (*in module ubelt.util_dict*), 40
dict_take() (*in module ubelt*), 135
dict_union() (*in module ubelt*), 136

dict_union() (*in module ubelt.util_dict*), 40
difference() (*ubelt.OrderedSet method*), 109
difference() (*ubelt.orderedset.OrderedSet method*), 11
difference_update() (*ubelt.OrderedSet method*), 110
difference_update() (*ubelt.orderedset.OrderedSet method*), 12
discard() (*ubelt.OrderedSet method*), 109
discard() (*ubelt.orderedset.OrderedSet method*), 11
display_message() (*ubelt.ProgIter method*), 115
display_message() (*ubelt.progiter.ProgIter method*), 18
download() (*in module ubelt*), 136
download() (*in module ubelt.util_download*), 46
dzip() (*in module ubelt*), 138
dzip() (*in module ubelt.util_dict*), 38

E

editfile() (*in module ubelt*), 139
encoding() (*ubelt.TeeStringIO property*), 116
encoding() (*ubelt.util_stream.TeeStringIO property*), 95
end() (*ubelt.ProgIter method*), 114
end() (*ubelt.progiter.ProgIter method*), 17
ensure() (*ubelt.Cacher method*), 103
ensure() (*ubelt.TempDir method*), 116
ensure() (*ubelt.util_cache.Cacher method*), 30
ensure() (*ubelt.util_path.TempDir method*), 83
ensure_app_cache_dir() (*in module ubelt*), 139
ensure_app_cache_dir() (*in module ubelt.util_platform*), 89
ensure_app_config_dir() (*in module ubelt*), 139
ensure_app_config_dir() (*in module ubelt.util_platform*), 88
ensure_app_data_dir() (*in module ubelt*), 140
ensure_app_data_dir() (*in module ubelt.util_platform*), 88
ensure_app_resource_dir() (*in module ubelt*), 140
ensure_newline() (*ubelt.ProgIter method*), 114
ensure_newline() (*ubelt.progiter.ProgIter method*), 18
ensure_unicode() (*in module ubelt*), 140
ensure_unicode() (*in module ubelt.util_str*), 93
ensuredir() (*in module ubelt*), 141
ensuredir() (*in module ubelt.util_path*), 85
existing_versions() (*ubelt.Cacher method*), 101
existing_versions() (*ubelt.util_cache.Cacher method*), 29
exists() (*ubelt.Cacher method*), 101
exists() (*ubelt.util_cache.Cacher method*), 29
expandpath() (*in module ubelt*), 141
expandpath() (*in module ubelt.util_path*), 86
expired() (*ubelt.CacheStamp method*), 99
expired() (*ubelt.util_cache.CacheStamp method*), 32

F

fileno() (*ubelt.TeeStringIO method*), 115
fileno() (*ubelt.util_stream.TeeStringIO method*), 94
find_duplicates() (*in module ubelt*), 142
find_duplicates() (*in module ubelt.util_dict*), 42
find_exe() (*in module ubelt*), 143
find_exe() (*in module ubelt.util_platform*), 89
find_path() (*in module ubelt*), 144
find_path() (*in module ubelt.util_platform*), 90
flatten() (*in module ubelt*), 145
flatten() (*in module ubelt.util_list*), 72
flush() (*ubelt.TeeStringIO method*), 116
flush() (*ubelt.util_stream.TeeStringIO method*), 95
FORCE_DISABLE (*ubelt.Cacher attribute*), 101
FORCE_DISABLE (*ubelt.util_cache.Cacher attribute*), 28
format_message() (*ubelt.ProgIter method*), 114
format_message() (*ubelt.progiter.ProgIter method*), 17
FormatterExtensions (*class in ubelt*), 104
FormatterExtensions (*class in ubelt.util_format*), 53

G

get_app_cache_dir() (*in module ubelt*), 145
get_app_cache_dir() (*in module ubelt.util_platform*), 89
get_app_config_dir() (*in module ubelt*), 145
get_app_config_dir() (*in module ubelt.util_platform*), 88
get_app_data_dir() (*in module ubelt*), 146
get_app_data_dir() (*in module ubelt.util_platform*), 87
get_app_resource_dir() (*in module ubelt*), 146
get_fpath() (*ubelt.Cacher method*), 101
get_fpath() (*ubelt.util_cache.Cacher method*), 28
get_indexer() (*ubelt.OrderedSet method*), 108
get_indexer() (*ubelt.orderedset.OrderedSet method*), 10
get_loc() (*ubelt.OrderedSet method*), 108
get_loc() (*ubelt.orderedset.OrderedSet method*), 10
grabdata() (*in module ubelt*), 146
grabdata() (*in module ubelt.util_download*), 48
group_items() (*in module ubelt*), 147
group_items() (*in module ubelt.util_dict*), 43

H

hash_data() (*in module ubelt*), 148
hash_data() (*in module ubelt.util_hash*), 56
hash_file() (*in module ubelt*), 149
hash_file() (*in module ubelt.util_hash*), 57

highlight_code() (*in module ubelt*), 151
highlight_code() (*in module ubelt.util_colors*), 36
hzcat() (*in module ubelt*), 151
hzcat() (*in module ubelt.util_str*), 93

I

identity() (*in module ubelt*), 152
identity() (*in module ubelt.util_func*), 54
import_module_from_name() (*in module ubelt*), 152
import_module_from_name() (*in module ubelt.util_import*), 61
import_module_from_path() (*in module ubelt*), 153
import_module_from_path() (*in module ubelt.util_import*), 62
indent() (*in module ubelt*), 154
indent() (*in module ubelt.util_str*), 91
index() (*ubelt.OrderedSet method*), 108
index() (*ubelt.orderedset.OrderedSet method*), 10
inject_method() (*in module ubelt*), 155
inject_method() (*in module ubelt.util_func*), 54
intersection() (*ubelt.OrderedSet method*), 109
intersection() (*ubelt.orderedset.OrderedSet method*), 11
intersection_update() (*ubelt.OrderedSet method*), 111
intersection_update() (*ubelt.orderedset.OrderedSet method*), 13
invert_dict() (*in module ubelt*), 155
invert_dict() (*in module ubelt.util_dict*), 43
isatty() (*ubelt.TeeStringIO method*), 115
isatty() (*ubelt.util_stream.TeeStringIO method*), 94
issubset() (*ubelt.OrderedSet method*), 110
issubset() (*ubelt.orderedset.OrderedSet method*), 12
issuperset() (*ubelt.OrderedSet method*), 110
issuperset() (*ubelt.orderedset.OrderedSet method*), 12
iter_window() (*in module ubelt*), 156
iter_window() (*in module ubelt.util_list*), 74
iterable() (*in module ubelt*), 157
iterable() (*in module ubelt.util_list*), 70
iteritems() (*in module ubelt.util_format*), 50

L

load() (*ubelt.Cacher method*), 102
load() (*ubelt.util_cache.Cacher method*), 30
log_part() (*ubelt.CaptureStdout method*), 104
log_part() (*ubelt.util_stream.CaptureStdout method*), 96
lookup() (*ubelt.FormatterExtensions method*), 105
lookup() (*ubelt.util_format.FormatterExtensions method*), 54

M

map_keys() (*in module ubelt*), 158
map_keys() (*in module ubelt.util_dict*), 44
map_vals() (*in module ubelt*), 158
map_vals() (*in module ubelt.util_dict*), 44
mean() (*ubelt.Timerit method*), 120
mean() (*ubelt.timerit.Timerit method*), 23
memoize() (*in module ubelt*), 159
memoize() (*in module ubelt.util_memoize*), 78
memoize_method (*class in ubelt*), 159
memoize_method (*class in ubelt.util_memoize*), 79
memoize_property() (*in module ubelt*), 160
memoize_property() (*in module ubelt.util_memoize*), 80
min() (*ubelt.Timerit method*), 119
min() (*ubelt.timerit.Timerit method*), 22
modname_to_modpath() (*in module ubelt*), 161
modname_to_modpath() (*in module ubelt.util_import*), 60
modpath_to_modname() (*in module ubelt*), 162
modpath_to_modname() (*in module ubelt.util_import*), 60
module
    ubelt, 97
    ubelt.__init__, 1
    ubelt.orderedset, 8
    ubelt.progiter, 13
    ubelt.timerit, 19
    ubelt.util_arg, 24
    ubelt.util_cache, 26
    ubelt.util_cmd, 33
    ubelt.util_colors, 35
    ubelt.util_const, 37
    ubelt.util_dict, 37
    ubelt.util_download, 46
    ubelt.util_format, 50
    ubelt.util_func, 54
    ubelt.util_hash, 55
    ubelt.util_import, 59
    ubelt.util_io, 64
    ubelt.util_links, 66
    ubelt.util_list, 68
    ubelt.util_memoize, 77
    ubelt.util_mixins, 81
    ubelt.util_path, 83
    ubelt.util_platform, 87
    ubelt.util_str, 91
    ubelt.util_stream, 94
    ubelt.util_time, 96

N

NiceRepr (*class in ubelt*), 105
NiceRepr (*class in ubelt.util_mixins*), 82
noborder() (*ubelt.chunks static method*), 128

noborder() (*ubelt.util_list.chunks static method*), 70

O

odict (*in module ubelt*), 163
odict (*in module ubelt.util_dict*), 46
OrderedSet (*class in ubelt*), 106
OrderedSet (*class in ubelt.orderedset*), 8
oset (*in module ubelt*), 163
oset (*in module ubelt.orderedset*), 13

P

paragraph() (*in module ubelt*), 163
paragraph() (*in module ubelt.util_str*), 92
peek() (*in module ubelt*), 163
peek() (*in module ubelt.util_list*), 77
platform_cache_dir() (*in module ubelt*), 164
platform_cache_dir() (*in module ubelt.util_platform*), 87
platform_config_dir() (*in module ubelt*), 164
platform_config_dir() (*in module ubelt.util_platform*), 87
platform_data_dir() (*in module ubelt*), 164
platform_data_dir() (*in module ubelt.util_platform*), 87
platform_resource_dir() (*in module ubelt*), 164
pop() (*ubelt.OrderedSet method*), 108
pop() (*ubelt.orderedset.OrderedSet method*), 10
print() (*ubelt.Timerit method*), 121
print() (*ubelt.timerit.Timerit method*), 24
ProgIter (*class in ubelt*), 111
ProgIter (*class in ubelt.progiter*), 15

R

rankings() (*ubelt.Timerit property*), 119
rankings() (*ubelt.timerit.Timerit property*), 22
readfrom() (*in module ubelt*), 165
readfrom() (*in module ubelt.util_io*), 64
register() (*ubelt.FormatterExtensions method*), 105
register() (*ubelt.util_format.FormatterExtensions method*), 53
renew() (*ubelt.CacheStamp method*), 99
renew() (*ubelt.util_cache.CacheStamp method*), 32
replicate() (*ubelt.chunks static method*), 128
replicate() (*ubelt.util_list.chunks static method*), 70
report() (*ubelt.Timerit method*), 120
report() (*ubelt.timerit.Timerit method*), 23
repr2() (*in module ubelt*), 165
repr2() (*in module ubelt.util_format*), 50
reset() (*ubelt.Timerit method*), 118
reset() (*ubelt.timerit.Timerit method*), 21

S

save() (*ubelt.Cacher method*), 102
save() (*ubelt.util_cache.Cacher method*), 30
set_extra() (*ubelt.ProgIter method*), 113
set_extra() (*ubelt.progiter.ProgIter method*), 16
shrinkuser() (*in module ubelt*), 167
shrinkuser() (*in module ubelt.util_path*), 85
sorted_keys() (*in module ubelt*), 168
sorted_keys() (*in module ubelt.util_dict*), 45
sorted_vals() (*in module ubelt*), 168
sorted_vals() (*in module ubelt.util_dict*), 45
split_modpath() (*in module ubelt*), 169
split_modpath() (*in module ubelt.util_import*), 59
start() (*ubelt.CaptureStdout method*), 104
start() (*ubelt.TempDir method*), 116
start() (*ubelt.util_path.TempDir method*), 83
start() (*ubelt.util_stream.CaptureStdout method*), 96
startfile() (*in module ubelt*), 169
std() (*ubelt.Timerit method*), 120
std() (*ubelt.timerit.Timerit method*), 23
step() (*ubelt.ProgIter method*), 113
step() (*ubelt.progiter.ProgIter method*), 17
stop() (*ubelt.CaptureStdout method*), 104
stop() (*ubelt.util_stream.CaptureStdout method*), 96
symlink() (*in module ubelt*), 170
symlink() (*in module ubelt.util_links*), 67
symmetric_difference() (*ubelt.OrderedSet method*), 110
symmetric_difference() (*ubelt.orderedset.OrderedSet method*), 12
symmetric_difference_update() (*ubelt.OrderedSet method*), 111
symmetric_difference_update() (*ubelt.orderedset.OrderedSet method*), 13

T

take() (*in module ubelt*), 171
take() (*in module ubelt.util_list*), 70
TeeStringIO (*class in ubelt*), 115
TeeStringIO (*class in ubelt.util_stream*), 94
TempDir (*class in ubelt*), 116
TempDir (*class in ubelt.util_path*), 83
tic() (*ubelt.Timer method*), 117
tic() (*ubelt.timerit.Timer method*), 20
Timer (*class in ubelt*), 116
Timer (*class in ubelt.timerit*), 19
Timerit (*class in ubelt*), 117
Timerit (*class in ubelt.timerit*), 20
timestamp() (*in module ubelt*), 172
timestamp() (*in module ubelt.util_time*), 96
to_dict() (*ubelt.AutoDict method*), 97
to_dict() (*ubelt.util_dict.AutoDict method*), 38
toc() (*ubelt.Timer method*), 117
toc() (*ubelt.timerit.Timer method*), 20
touch() (*in module ubelt*), 172
touch() (*in module ubelt.util_io*), 65

truepath() (*in module ubelt*), 173
tryload() (*ubelt.Cacher method*), 102
tryload() (*ubelt.util_cache.Cacher method*), 29

U

ubelt
    module, 97
ubelt.__init__
    module, 1
ubelt.orderedset
    module, 8
ubelt.progiter
    module, 13
ubelt.timerit
    module, 19
ubelt.util_arg
    module, 24
ubelt.util_cache
    module, 26
ubelt.util_cmd
    module, 33
ubelt.util_colors
    module, 35
ubelt.util_const
    module, 37
ubelt.util_dict
    module, 37
ubelt.util_download
    module, 46
ubelt.util_format
    module, 50
ubelt.util_func
    module, 54
ubelt.util_hash
    module, 55
ubelt.util_import
    module, 59
ubelt.util_io
    module, 64
ubelt.util_links
    module, 66
ubelt.util_list
    module, 68
ubelt.util_memoize
    module, 77
ubelt.util_mixins
    module, 81
ubelt.util_path
    module, 83
ubelt.util_platform
    module, 87
ubelt.util_str
    module, 91
ubelt.util_stream
    module, 94
ubelt.util_time
    module, 96
union() (*ubelt.OrderedSet method*), 109
union() (*ubelt.orderedset.OrderedSet method*), 11
unique() (*in module ubelt*), 173
unique() (*in module ubelt.util_list*), 72
unique_flags() (*in module ubelt*), 174
unique_flags() (*in module ubelt.util_list*), 73
update() (*ubelt.OrderedSet method*), 107
update() (*ubelt.orderedset.OrderedSet method*), 9
userhome() (*in module ubelt*), 174
userhome() (*in module ubelt.util_path*), 85

V

VERBOSE (*ubelt.Cacher attribute*), 101
VERBOSE (*ubelt.util_cache.Cacher attribute*), 28

W

write() (*ubelt.TeeStringIO method*), 116
write() (*ubelt.util_stream.TeeStringIO method*), 95
writeto() (*in module ubelt*), 175
writeto() (*in module ubelt.util_io*), 64