# Contents

1 Installation and Integration 3

2 Third-party packages 5

3 Gaps 7

4 Section listing 9

1. Architecture ................................................. 9
2. cacheutils - Caches and caching ............................ 10
3. debugutils - Debugging utilities ............................ 15
4. dictutils - Mapping types (OMD) ............................ 16
5. ecoutils - Ecosystem analytics ............................ 22
6. fileutils - Filesystem helpers ............................ 24
7. formatutils - str.format() toolbox .......................... 27
8. funcutils - functools fixes .................................. 29
9. gcutils - Garbage collecting tools .......................... 34
10. ioutils - Input/output enhancements ........................ 35
11. iterutils - itertools improvements ........................ 37
12. jsonutils - JSON interactions ............................ 47
13. listutils - list derivatives .................................. 48
14. mathutils - Mathematical functions ........................ 49
15. mboxutils - Unix mailbox utilities ........................ 51
16. namedutils - Lightweight containers ...................... 51
17. pathutils - Filesystem fun .................................. 52
18. queueutils - Priority queues .............................. 54
19. setutils - IndexedSet type .................................. 55
20. socketutils - socket wrappers .............................. 58
21. statsutils - Statistics fundamentals ....................... 62
22. strutils - Text manipulation .............................. 69
23. tableutils - 2D data structure .............................. 74
24. tbutils - Tracebacks and call stacks ....................... 76
25. timeutils - datetime additions ............................. 80
26. typeutils - Type handling .................................. 84
27. urlutils - Structured URL ................................. 85

Python Module Index 93
boltons should be builtins.

Boltons is a set of pure-Python utilities in the same spirit as — and yet conspicuously missing from — the standard library, including:

- Atomic file saving, bolted on with fileutils
- A highly-optimized OrderedMultiDict, in dictutils
- Two types of PriorityQueue, in queueutils
- Chunked and windowed iteration, in iterutils
- A full-featured TracebackInfo type, for representing stack traces, in tbutils
- A lightweight UTC timezone available in timeutils.
- Recursive mapping for nested data transforms, with remap

And that’s just a small selection. As of March 30, 2020, boltons is 83 types and 160 functions, spread across 29 modules. See them all in the genindex, and see what’s new by checking the CHANGELOG.
CHAPTER 1

Installation and Integration

Boltons can be added to a project in a few ways. There’s the obvious one:

```
pip install boltons
```

Then dozens of boltons are just an import away:

```
from boltons.cacheutils import LRU
lru_cache = LRU()
lru_cache['result'] = 'success'
```

Due to the nature of utilities, application developers might want to consider other integration options. See the Integration section of the architecture document for more details.

Boltons is tested against Python 2.6-2.7, 3.4-3.7, and PyPy.
The majority of boltons strive to be “good enough” for a wide range of basic uses, leaving advanced use cases to Python’s myriad specialized 3rd-party libraries. In many cases the respective boltons module will describe 3rd-party alternatives worth investigating when use cases outgrow boltons. If you’ve found a natural “next-step” library worth mentioning, consider filing an issue!
Found something missing in the standard library that should be in `boltons`? Found something missing in `boltons`? First, take a moment to read the very brief _Architecture_ statement to make sure the functionality would be a good fit. Then, if you are very motivated, submit a Pull Request. Otherwise, submit a short feature request on the Issues page, and we will figure something out.
4.1 Architecture

`boltons` has a minimalist architecture: remain as consistent, and self-contained as possible, with an eye toward maintaining its range of use cases and usage patterns as wide as possible.

4.1.1 Integration

Utility libraries are often used extensively within a project, and because they are not often fundamental to the architecture of the application, simplicity and stability may take precedence over version recency. In these cases, developers can:

1. Copy the whole `boltons` package into a project.
2. Copy just the `utils.py` file that a project requires.

`Boltons` take this into account by design. The `boltons` package depends on no packages, making it easy for inclusion into a project. Furthermore, virtually all individual modules have been written to be as self-contained as possible, allowing cherrypicking of functionality into projects.

4.1.2 Design of a `bolton`

`boltons` aims to be a living library, an ever-expanding collection of tested and true utilities. For a `bolton` to be a `bolton`, it should:

1. Be pure-Python and as self-contained as possible.
2. Perform a common task or fulfill a common role.
3. Demonstrate and mitigate some insufficiency in the standard library.
4. Strive for the standard set forth by the standard library by striking a balance between best practice and “good enough”, correctness and common sense. When in doubt, ask, “what would the standard library do?”
5. Have approachable documentation with at least one helpful doctest, links to relevant standard library functionality, as well as any 3rd-party packages that provide further capabilities.

Finally, boltons should be substantial implementations of commonly trivialized stumbling blocks and not the other way around. The larger the problem solved, the less likely the functionality is suitable for inclusion in boltons; boltons are fundamental and self-contained, not sweeping and architecture-defining.

### 4.1.3 Themes of boltons

boltons has had a wide variety of inspirations over the years, but a definite set of themes have emerged:

1. From the Python docs:
   1. `queueutils` - `heapq` docs
   2. `iterutils` - `itertools` docs
   3. `timeutils` - `datetime` docs

2. Reimplementations and tweaks of the standard library:
   1. `boltons.fileutils.copytree()` - `shutil.copytree()`
   2. `boltons.namedutils.namedtuple` - `collections.namedtuple()`

3. One-off implementations discovered in multiple other libraries’ `utils.py` or equivalent
   1. `boltons.strutils.slugify()`
   2. `boltons.strutils.bytes2human()`
   3. `boltons.timeutils.relative_time()`

4. More powerful multi-purpose data structures
   1. `boltons.dictutils.OrderedMultiDict`
   2. `boltons.setutils.IndexedSet`
   3. `boltons.listutils.BList`
   4. `boltons.namedutils.namedlist`
   5. `boltons.tableutils.Table`

5. Personal practice and experience
   1. `boltons.debugutils`
   2. `boltons.gcutils`
   3. `boltons.tbutils`

### 4.2 cacheutils - Caches and caching

`cacheutils` contains consistent implementations of fundamental cache types. Currently there are two to choose from:

- **LRI** - Least-recently inserted
- **LRU** - Least-recently used
Both caches are `dict` subtypes, designed to be as interchangeable as possible, to facilitate experimentation. A key practice with performance enhancement with caching is ensuring that the caching strategy is working. If the cache is constantly missing, it is just adding more overhead and code complexity. The standard statistics are:

- `hit_count` - the number of times the queried key has been in the cache
- `miss_count` - the number of times a key has been absent and/or fetched by the cache
- `soft_miss_count` - the number of times a key has been absent, but a default has been provided by the caller, as with `dict.get()` and `dict.setdefault()`. Soft misses are a subset of misses, so this number is always less than or equal to `miss_count`.

Additionally, `cacheutils` provides `ThresholdCounter`, a cache-like bounded counter useful for online statistics collection.

Learn more about caching algorithms on Wikipedia.

### 4.2.1 Least-Recently Inserted (LRI)

The `LRI` is the simpler cache, implementing a very simple first-in, first-out (FIFO) approach to cache eviction. If the use case calls for simple, very-low overhead caching, such as somewhat expensive local operations (e.g., string operations), then the LRI is likely the right choice.

```python
import boltons.cacheutils

class boltons.cacheutils.LRI(max_size=128, values=None, on_miss=None)
    The LRI implements the basic Least Recently Inserted strategy to caching. One could also think of this as a SizeLimitedDefaultDict.

    on_miss is a callable that accepts the missing key (as opposed to collections.defaultdict's "default_factory", which accepts no arguments.) Also note that, like the LRI, the LRI is instrumented with statistics tracking.

>>> cap_cache = LRI(max_size=2)
>>> cap_cache['a'], cap_cache['b'] = 'A', 'B'
>>> from pprint import pprint as pp
>>> pp(dict(cap_cache))
{'a': 'A', 'b': 'B'}
>>> [cap_cache['b'] for i in range(3)][0]
'B'
>>> cap_cache['c'] = 'C'
>>> print(cap_cache.get('a'))
None
>>> cap_cache.hit_count, cap_cache.miss_count, cap_cache.soft_miss_count
(3, 1, 1)
```

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

copy() → a shallow copy of D

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

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

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

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

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

The **LRU** is the more advanced cache, but it’s still quite simple. When it reaches capacity, a new insertion replaces the least-recently used item. This strategy makes the LRU a more effective cache than the LRI for a wide variety of applications, but also entails more operations for all of its APIs, especially reads. Unlike the **LRI**, the LRU has thread safety built in.

```python
class boltons.cacheutils.LRU(max_size=128, values=None, on_miss=None)
```

The LRU is **dict** subtype implementation of the **Least-Recently Used** caching strategy.

**Parameters**

- `max_size (int)` – Max number of items to cache. Defaults to 128.
- `values (iterable)` – Initial values for the cache. Defaults to None.
- `on_miss (callable)` – a callable which accepts a single argument, the key not present in the cache, and returns the value to be cached.

```python
>>> cap_cache = LRU(max_size=2)
>>> cap_cache['a'], cap_cache['b'] = 'A', 'B'
>>> from pprint import pprint as pp
>>> pp(dict(cap_cache))
{'a': 'A', 'b': 'B'}
>>> [cap_cache['b'] for i in range(3)][0]
'B'
>>> cap_cache['c'] = 'C'
>>> print(cap_cache.get('a'))
None
```

This cache is also instrumented with statistics collection. **hit_count**, **miss_count**, and **soft_miss_count** are all integer members that can be used to introspect the performance of the cache. (“Soft” misses are misses that did not raise **KeyError**, e.g., LRU.get() or on_miss was used to cache a default.

```python
>>> cap_cache.hit_count, cap_cache.miss_count, cap_cache.soft_miss_count
(3, 1, 1)
```

Other than the size-limiting caching behavior and statistics, LRU acts like its parent class, the built-in Python **dict**.

4.2.3 Automatic function caching

Continuing in the theme of cache tunability and experimentation, **cacheutils** also offers a pluggable way to cache function return values: the **cached()** function decorator and the **cachedmethod()** method decorator.

```python
boltons.cacheutils.cached(cache, scoped=True, typed=False, key=None)
```

Cache any function with the cache object of your choosing. Note that the function wrapped should take only **hashable** arguments.

**Parameters**

- `cache (Mapping)` – Any **dict**-like object suitable for use as a cache. Instances of the **LRU** and **LRI** are good choices, but a plain **dict** can work in some cases, as well. This argument can also be a callable which accepts no arguments and returns a mapping.
- `scoped (bool)` – Whether the function itself is part of the cache key. **True** by default, different functions will not read one another’s cache entries, but can evict one another’s
results. False can be useful for certain shared cache use cases. More advanced behavior can be produced through the key argument.

- **typed (bool)** – Whether to factor argument types into the cache check. Default False, setting to True causes the cache keys for 3 and 3.0 to be considered unequal.

```python
>>> my_cache = LRU()
>>> @cached(my_cache)
... def cached_lower(x):
...     return x.lower()
...
>>> cached_lower("CaChInG's FuN AgAiN!")
"caching's fun again!"
>>> len(my_cache)
1
```

boltons.cacheutils.cachedmethod(cache, scoped=True, typed=False, key=None)

Similar to cached(), cachedmethod is used to cache methods based on their arguments, using any dict-like cache object.

Parameters

- **cache (str/Mapping/callable)** – Can be the name of an attribute on the instance, any Mapping/dict-like object, or a callable which returns a Mapping.

- **scoped (bool)** – Whether the method itself and the object it is bound to are part of the cache keys. True by default, different methods will not read one another’s cache results. False can be useful for certain shared cache use cases. More advanced behavior can be produced through the key arguments.

- **typed (bool)** – Whether to factor argument types into the cache check. Default False, setting to True causes the cache keys for 3 and 3.0 to be considered unequal.

- **key (callable)** – A callable with a signature that matches make_cache_key() that returns a tuple of hashable values to be used as the key in the cache.

```python
>>> class Lowerer(object):
...     def __init__(self):
...         self.cache = LRU()
...
...     @cachedmethod('cache')
...     def lower(self, text):
...         return text.lower()
...
>>> lowerer = Lowerer()
>>> lowerer.lower('WOW WHO COULD GUESS CACHING COULD BE SO NEAT')
'wow who could guess caching could be so neat'
>>> len(lowerer.cache)
1
```

Similar functionality can be found in Python 3.4’s functools.lru_cache() decorator, but the functools approach does not support the same cache strategy modification, nor does it support sharing the cache object across multiple functions.

boltons.cacheutils.cachedproperty(func)

The cachedproperty is used similar to property, except that the wrapped method is only called once. This is commonly used to implement lazy attributes.

After the property has been accessed, the value is stored on the instance itself, using the same name as the cachedproperty. This allows the cache to be cleared with delattr(), or through manipulating the object’s
__dict__.

4.2.4 Threshold-bounded Counting

class boltons.cacheutils.ThresholdCounter (threshold=0.001)

A bounded dict-like Mapping from keys to counts. The ThresholdCounter automatically compacts after every 
(1 / threshold) additions, maintaining exact counts for any keys whose count represents at least a threshold ratio 
of the total data. In other words, if a particular key is not present in the ThresholdCounter, its count represents 
less than threshold of the total data.

```python
>>> tc = ThresholdCounter(threshold=0.1)
>>> tc.add(1)
>>> tc.items()
[(1, 1)]
>>> tc.update([2] * 10)
>>> tc.get(1)
0
>>> tc.add(5)
>>> 5 in tc
True
>>> len(list(tc.elements()))
11
```

As you can see above, the API is kept similar to collections.Counter. The most notable feature omissions being that counted items cannot be set directly, uncounted, or removed, as this would disrupt the math.

Use the ThresholdCounter when you need best-effort long-lived counts for dynamically-keyed data. Without 
a bounded datastructure such as this one, the dynamic keys often represent a memory leak and can impact application reliability. The ThresholdCounter’s item replacement strategy is fully deterministic and can be thought of as Amortized Least Relevant. The absolute upper bound of keys it will store is (2/threshold), but realistically (1/threshold) is expected for uniformly random datastreams, and one or two orders of magnitude better for real-world data.

This algorithm is an implementation of the Lossy Counting algorithm described in “Approximate Frequency Counts over Data Streams” by Manku & Motwani. Hat tip to Kurt Rose for discovery and initial implementation.

add(key)
Increment the count of key by 1, automatically adding it if it does not exist.

Cache compaction is triggered every 1/threshold additions.

elements()
Return an iterator of all the common elements tracked by the counter. Yields each key as many times as it has been seen.

get(key, default=0)
Get count for key, defaulting to 0.

get_common_count()
Get the sum of counts for keys exceeding the configured data threshold.

get_commonality()
Get a float representation of the effective count accuracy. The higher the number, the less uniform the keys being added, and the higher accuracy and efficiency of the ThresholdCounter.

If a stronger measure of data cardinality is required, consider using hyperloglog.
get_uncommon_count()
Get the sum of counts for keys that were culled because the associated counts represented less than the
configured threshold. The long-tail counts.

most_common(n=None)
Get the top n keys and counts as tuples. If n is omitted, returns all the pairs.

update(iterable, **kwargs)
Like dict.update() but add counts instead of replacing them, used to add multiple items in one call.
Source can be an iterable of keys to add, or a mapping of keys to integer counts.

4.3 debugutils - Debugging utilities

A small set of utilities useful for debugging misbehaving applications. Currently this focuses on ways to use pdb, the
built-in Python debugger.

boltons.debugutils.pdb_on_signal(signalnum=None)
Installs a signal handler for signalnum, which defaults to SIGINT, or keyboard interrupt/ctrl-c. This signal
handler launches a pdb breakpoint. Results vary in concurrent systems, but this technique can be useful for
debugging infinite loops, or easily getting into deep call stacks.

Parameters signalnum (int) – The signal number of the signal to handle with pdb. Defaults to
signal.SIGINT, see signal for more information.

boltons.debugutils.pdb_on_exception(limit=100)
Installs a handler which, instead of exiting, attaches a post-mortem pdb console whenever an unhandled excep-
tion is encountered.

Parameters limit (int) – the max number of stack frames to display when printing the traceback

A similar effect can be achieved from the command-line using the following command:

```
python -m pdb your_code.py
```

But pdb_on_exception allows you to do this conditionally and within your application. To restore default
behavior, just do:

```
sys.excepthook = sys.__excepthook__
```

boltons.debugutils.wrap_trace(obj, hook=<function trace_print_hook>, which=None,
events=None, label=None)
Monitor an object for interactions. Whenever code calls a method, gets an attribute, or sets an attribute, an event
is called. By default the trace output is printed, but a custom tracing hook can be passed.

Parameters

- obj (object) – New- or old-style object to be traced. Built-in objects like lists and dicts
also supported.
- hook (callable) – A function called once for every event. See below for details.
- which (str) – One or more attribute names to trace, or a function accepting attribute name
and value, and returning True/False.
- events (str) – One or more kinds of events to call hook on. Expected values
are ['get', 'set', 'del', 'call', 'raise', 'return']. Defaults to all
events.
• **label**(str) – A name to associate with the traced object. Defaults to hexadecimal memory address, similar to `repr`.

The object returned is not the same object as the one passed in. It will not pass identity checks. However, it will pass `isinstance()` checks, as it is a new instance of a new subtype of the object passed.

## 4.4 dictutils - Mapping types (OMD)

Python has a very powerful mapping type at its core: the `dict` type. While versatile and featureful, the `dict` prioritizes simplicity and performance. As a result, it does not retain the order of item insertion\(^1\), nor does it store multiple values per key. It is a fast, unordered 1:1 mapping.

The `OrderedMultiDict` contrasts to the built-in `dict`, as a relatively maximalist, ordered 1:n subtype of `dict`. Virtually every feature of `dict` has been retooled to be intuitive in the face of this added complexity. Additional methods have been added, such as `collections.Counter`-like functionality.

A prime advantage of the `OrderedMultiDict` (OMD) is its non-destructive nature. Data can be added to an OMD without being rearranged or overwritten. The property can allow the developer to work more freely with the data, as well as make more assumptions about where input data will end up in the output, all without any extra work.

One great example of this is the `OMD.inverted()` method, which returns a new OMD with the values as keys and the keys as values. All the data and the respective order is still represented in the inverted form, all from an operation which would be outright wrong and reckless with a built-in `dict` or `collections.OrderedDict`.

The OMD has been performance tuned to be suitable for a wide range of usages, including as a basic unordered MultiDict. Special thanks to Mark Williams for all his help.

```python
boltons.dictutils.MultiDict
alias of boltons.dictutils.OrderedMultiDict

boltons.dictutils.OMD
alias of boltons.dictutils.OrderedMultiDict

class boltons.dictutils.OrderedMultiDict(*args, **kwargs)

A MultiDict is a dictionary that can have multiple values per key and the `OrderedMultiDict` (OMD) is a MultiDict that retains original insertion order. Common use cases include:

• handling query strings parsed from URLs

• inverting a dictionary to create a reverse index (values to keys)

• stacking data from multiple dictionaries in a non-destructive way

The `OrderedMultiDict` constructor is identical to the built-in `dict`, and overall the API constitutes an intuitive superset of the built-in type:

```python
>>> omd = OrderedMultiDict()
>>> omd['a'] = 1
>>> omd['b'] = 2
>>> omd.add('a', 3)
>>> omd.get('a')
3
>>> omd.getlist('a')
[1, 3]
```

Some non-`dict`-like behaviors also make an appearance, such as support for `reversed()`:

\(^1\) As of 2015, basic dicts on PyPy are ordered, and as of December 2017, basic dicts in CPython 3 are now ordered, as well.
Note that unlike some other MultiDicts, this OMD gives precedence to the most recent value added. `omd['a']` refers to 3, not 1.

```python
>>> omd
OrderedMultiDict([('a', 1), ('b', 2), ('a', 3)])
```

```python
>>> omd.poplast('a')
3
>>> omd
OrderedMultiDict([('a', 1), ('b', 2)])
```

```python
>>> omd.pop('a')
1
>>> omd
OrderedMultiDict([('b', 2)])
```

If you want a safe-to-modify or flat dictionary, use `OrderedMultiDict.todict()`.

```python
>>> from pprint import pprint as pp
# preserve printed ordering
>>> omd = OrderedMultiDict([('a', 1), ('b', 2), ('a', 3)])
```

```python
>>> pp(omd.todict())
{'a': 1, 'b': 2}
```

```python
>>> pp(omd.todict(multi=True))
{'a': [1, 3], 'b': [2]}
```

With `multi=False`, items appear with the keys in to original insertion order, alongside the most-recently inserted value for that key.

```python
>>> OrderedMultiDict([('a', 1), ('b', 2), ('a', 3)]).items(multi=False)
[('a', 3), ('b', 2)]
```

**Warning:** `dict(omd)` changed behavior in Python 3.7 due to changes made to support the transition from `collections.OrderedDict` to the built-in dictionary being ordered. Before 3.7, the result would be a new dictionary, with values that were lists, similar to `omd.todict(multi=True)` (but only shallow-copy; the lists were direct references to OMD internal structures). From 3.7 onward, the values became singular, like `omd.todict(multi=False)`. For reliable cross-version behavior, just use `todict()`.

### add

Add a single value `v` under a key `k`. Existing values under `k` are preserved.

### addlist

Add an iterable of values underneath a specific key, preserving any values already under that key.

```python
>>> omd = OrderedMultiDict([('a', -1)])
>>> omd.addlist('a', range(3))
>>> omd
OrderedMultiDict([('a', -1), ('a', 0), ('a', 1), ('a', 2)])
```

Called `addlist` for consistency with `getlist()`, but tuples and other sequences and iterables work.

### clear

Empty the dictionary.

### copy

Return a shallow copy of the dictionary.
counts()
Returns a mapping from key to number of values inserted under that key. Like collections.Counter, but returns a new OrderedMultiDict.

classmethod fromkeys(keys, default=None)
Create a dictionary from a list of keys, with all the values set to default, or None if default is not set.

get(k, default=None)
Return the value for key k if present in the dictionary, else default. If default is not given, None is returned. This method never raises a KeyError.

To get all values under a key, use OrderedMultiDict.getlist().

getlist(k, default=_MISSING)
Get all values for key k as a list, if k is in the dictionary, else default. The list returned is a copy and can be safely mutated. If default is not given, an empty list is returned.

inverted()
Returns a new OrderedMultiDict with values and keys swapped, like creating dictionary transposition or reverse index. Insertion order is retained and all keys and values are represented in the output.

```python
>>> omd = OMD([(0, 2), (1, 2)])
>>> omd.inverted().getlist(2)
[0, 1]
```

Inverting twice yields a copy of the original:

```python
>>> omd.inverted().inverted()
OrderedMultiDict([(0, 2), (1, 2)])
```

items(multi=False)
Returns a list containing the output of iteritems(). See that method’s docs for more details.

iteritems(multi=False)
Iterate over the OMD’s items in insertion order. By default, yields only the most-recently inserted value for each key. Set multi to True to get all inserted items.

iterkeys(multi=False)
Iterate over the OMD’s keys in insertion order. By default, yields each key once, according to the most recent insertion. Set multi to True to get all keys, including duplicates, in insertion order.

itervalues(multi=False)
Iterate over the OMD’s values in insertion order. By default, yields the most-recently inserted value per unique key. Set multi to True to get all values according to insertion order.

keys(multi=False)
Returns a list containing the output of iterkeys(). See that method’s docs for more details.

pop(k, default=_MISSING)
Remove all values under key k, returning the most-recently inserted value. Raises KeyError if the key is not present and no default is provided.

popall(k, default=_MISSING)
Remove all values under key k, returning them in the form of a list. Raises KeyError if the key is not present and no default is provided.

poplast(k=_MISSING, default=_MISSING)
Remove and return the most-recently inserted value under the key k, or the most-recently inserted key if k is not provided. If no values remain under k, it will be removed from the OMD. Raises KeyError if k is not present in the dictionary, or the dictionary is empty.
**setdefault** (*k, default=_MISSING*)

If key *k* is in the dictionary, return its value. If not, insert *k* with a value of *default* and return *default*. *default* defaults to *None*. See `dict.setdefault()` for more information.

**sorted** (*key=None, reverse=False*)

Similar to the built-in `sorted()`, except this method returns a new `OrderedMultiDict` sorted by the provided key function, optionally reversed.

**Parameters**

- **key** (*callable*) – A callable to determine the sort key of each element. The callable should expect an item (key-value pair tuple).
- **reverse** (*bool*) – Set to True to reverse the ordering.

```python
>>> omd = OrderedMultiDict(zip(range(3), range(3)))
>>> omd.sorted(reverse=True)
OrderedMultiDict([(2, 2), (1, 1), (0, 0)])
```

Note that the key function receives an item (key-value tuple), so the recommended signature looks like:

```python
>>> omd = OrderedMultiDict(zip('hello', 'world'))
>>> omd.sorted(key=lambda i: i[1])  # i[0] is the key, i[1] is the val
OrderedMultiDict([('o', 'd'), ('l', 'l'), ('e', 'o'), ('l', 'r'), ('h', 'w')])
```

**sortedvalues** (*key=None, reverse=False*)

Returns a copy of the `OrderedMultiDict` with the same keys in the same order as the original OMD, but the values within each keyspace have been sorted according to *key* and *reverse*.

**Parameters**

- **key** (*callable*) – A single-argument callable to determine the sort key of each element. The callable should expect an item (key-value pair tuple).
- **reverse** (*bool*) – Set to True to reverse the ordering.

```python
>>> omd = OrderedMultiDict()
>>> omd.addlist('even', [6, 2])
>>> omd.addlist('odd', [1, 5])
>>> omd.add('even', 4)
>>> omd.add('odd', 3)
>>> somd = omd.sortedvalues()
>>> somd.getlist('even')
[2, 4, 6]
>>> somd.keys(multi=True) == omd.keys(multi=True)
True
>>> omd == somd
False
>>> somd
OrderedMultiDict([('even', 2), ('even', 4), ('odd', 1), ('odd', 3), ('even', 6), ('odd', 5)])
```

As demonstrated above, contents and key order are retained. Only value order changes.

**todict** (*multi=False*)

Gets a basic `dict` of the items in this dictionary. Keys are the same as the OMD, values are the most recently inserted values for each key.

Setting the *multi* arg to True is yields the same result as calling `dict` on the OMD, except that all the value lists are copies that can be safely mutated.
update \((E, **F)\)
Add items from a dictionary or iterable (and/or keyword arguments), overwriting values under an existing key. See `dict.update()` for more details.

update_extend \((E, **F)\)
Add items from a dictionary, iterable, and/or keyword arguments without overwriting existing items present in the dictionary. Like `update()`, but adds to existing keys instead of overwriting them.

values \((multi=False)\)
Returns a list containing the output of `itervalues()`. See that method’s docs for more details.

viewitems () → a set-like object providing a view on OMD’s items
viewkeys () → a set-like object providing a view on OMD’s keys
viewvalues () → an object providing a view on OMD’s values

class boltons.dictutils.OneToOne (*a, **kw)
Implements a one-to-one mapping dictionary. In addition to inheriting from and behaving exactly like the builtin `dict`, all values are automatically added as keys on a reverse mapping, available as the `inv` attribute. This arrangement keeps key and value namespaces distinct.

Basic operations are intuitive:

```python
>>> oto = OneToOne({'a': 1, 'b': 2})
>>> print(oto['a'])
1
>>> print(oto.inv[1])
a
>>> len(oto)
2
```

Overwrites happens in both directions:

```python
>>> oto.inv[1] = 'c'
>>> print(oto.get('a'))
None
>>> len(oto)
2
```

For a very similar project, with even more one-to-one functionality, check out `bidict`.

clear () → None. Remove all items from D.
copy () → a shallow copy of D
ing

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

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

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

classmethod unique (*a, **kw)
This alternate constructor for OneToOne will raise an exception when input values overlap. For instance:

```python
>>> OneToOne.unique({'a': 1, 'b': 1})
Traceback (most recent call last):
```

(continues on next page)
This even works across inputs:

```python
>>> a_dict = {'a': 2}
>>> OneToOne.unique(a_dict, b=2)
Traceback (most recent call last):
  ...  
ValueError: expected unique values, got multiple keys for the following...
  →values: ...
```

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

class boltons.dictutils.ManyToMany(items=None)
a dict-like entity that represents a many-to-many relationship between two groups of objects
behaves like a dict-of-tuples; also has .inv which is kept up to date which is a dict-of-tuples in the other direction
also, can be used as a directed graph among hashable python objects

add(key, val)
get(key, default=frozenset([]))
iteritems()
keys()
remove(key, val)
replace(key, newkey)
replace instances of key by newkey
update(iterable)
given an iterable of (key, val), add them all

boltons.dictutils.subdict(d, keep=None, drop=None)
Compute the "subdictionary" of a dict, d.
A subdict is to a dict what a subset is to set. If A is a subdict of B, that means that all keys of A are present in B.
Returns a new dict with any keys in drop removed, and any keys in keep still present, provided they were in the original dict. keep defaults to all keys, drop defaults to empty, so without one of these arguments, calling this function is equivalent to calling dict().

```python
>>> from pprint import pprint as pp
>>> pp(subdict({"a": 1, "b": 2}))
{'a': 1, 'b': 2}
>>> pp(subdict({"a": 1, "b": 2, 'c': 3}, drop=['b', 'c']))
{'a': 1}
>>> pp(subdict({"a": 1, "b": 2, 'c': 3}, keep=['a', 'c']))
{'a': 1, 'c': 3}
```

class boltons.dictutils.FrozenDict
An immutable dict subtype that is hashable and can itself be used as a dict key or set entry. What frozenset is to set, FrozenDict is to dict.
There was once an attempt to introduce such a type to the standard library, but it was rejected: PEP 416. Because FrozenDict is a \texttt{dict} subtype, it automatically works everywhere a dict would, including JSON serialization.

\begin{verbatim}
clear (*a, **kw)
       raises a TypeError, because FrozenDicts are immutable

classmethod fromkeys \((S, v)\) \rightarrow \text{New dict with keys from } S \text{ and values equal to } v.
       \text{v defaults to None.}

pop (*a, **kw)
       raises a TypeError, because FrozenDicts are immutable

popitem (*a, **kw)
       raises a TypeError, because FrozenDicts are immutable

setdefault (*a, **kw)
       raises a TypeError, because FrozenDicts are immutable

update (*a, **kw)
       raises a TypeError, because FrozenDicts are immutable

updated (*a, **kw)
       Make a copy and add items from a dictionary or iterable (and/or keyword arguments), overwriting values
       under an existing key. See \texttt{dict.update()} for more details.
\end{verbatim}

\section{4.5 \texttt{ecoutils} - Ecosystem analytics}

As a programming ecosystem grows, so do the chances of runtime variability.

Python boasts one of the widest deployments for a high-level programming environment, making it a viable target for all manner of application. But with breadth comes variance, so it’s important to know what you’re working with.

Some basic variations that are common among development machines:

-\begin{itemize}
  \item **Executable runtime:** CPython, PyPy, Jython, etc., plus build date and compiler
  \item **Language version:** 2.4, 2.5, 2.6, 2.7… 3.4, 3.5, 3.6
  \item **Host operating system:** Windows, OS X, Ubuntu, Debian, CentOS, RHEL, etc.
  \item **Features:** 64-bit, IPv6, Unicode character support (UCS-2/UCS-4)
  \item **Built-in library support:** OpenSSL, threading, SQLite, zlib
  \item **User environment:** umask, ulimit, working directory path
  \item **Machine info:** CPU count, hostname, filesystem encoding
\end{itemize}

See the full example profile below for more.

ecoutils was created to quantify that variability. ecoutils quickly produces an information-dense description of critical runtime factors, with minimal side effects. In short, ecoutils is like browser and user agent analytics, but for Python environments.

\section{4.5.1 Transmission and collection}

The data is all JSON serializable, and is suitable for sending to a central analytics server. An HTTP-backed service for this can be found at: https://github.com/mahmoud/espymetrics/
4.5.2 Notable omissions

Due to space constraints (and possibly latency constraints), the following information is deemed not dense enough, and thus omitted:

- `sys.path`
- `full sysconfig`
- Environment variables (`os.environ`)

4.5.3 Compatibility

So far `ecoutils` has has been tested on Python 2.4, 2.5, 2.6, 2.7, 3.4, 3.5, and PyPy. Various versions have been tested on Ubuntu, Debian, RHEL, OS X, FreeBSD, and Windows 7.

**Note:** Boltons typically only support back to Python 2.6, but due to its nature, `ecoutils` extends backwards compatibility to Python 2.4 and 2.5.

4.5.4 Profile generation

Profiles are generated by `ecoutils.get_profile()`.

When run as a module, `ecoutils` will call `get_profile()` and print a profile in JSON format:

```bash
$ python -m boltons.ecoutils
{
    "_eco_version": "1.0.0",
    "cpu_count": 4,
    "cwd": "/home/mahmoud/projects/boltons",
    "fs_encoding": "UTF-8",
    "guid": "6b139e7bbf5ad4ed8d4063bf6235b4d2",
    "hostfqdn": "mahmoud-host",
    "hostname": "mahmoud-host",
    "linux_dist_name": "Ubuntu",
    "linux_dist_version": "14.04",
    "python": {
        "argv": "boltons/ecoutils.py",
        "bin": "/usr/bin/python",
        "build_date": "Jun 22 2015 17:58:13",
        "compiler": "GCC 4.8.2",
        "features": {
            "64bit": true,
            "expat": "expat_2.1.0",
            "ipv6": true,
            "openssl": "OpenSSL 1.0.1f 6 Jan 2014",
            "readline": true,
            "sqlite": "3.8.2",
            "threading": true,
            "tkinter": "8.6",
            "unicode_wide": true,
            "zlib": "1.2.8"
        },
        "version": "2.7.6 (default, Jun 22 2015, 17:58:13) [GCC 4.8.2]",
        "version_info": [
```
pip install boltons and try it yourself!

boltons.ecoutils.get_profile(**kwargs)

The main entrypoint to ecoutils. Calling this will return a JSON-serializable dictionary of information about the current process.

It is very unlikely that the information returned will change during the lifetime of the process, and in most cases the majority of the information stays the same between runs as well.

get_profile() takes one optional keyword argument, scrub, a bool that, if True, blanks out identifiable information. This includes current working directory, hostname, Python executable path, command-line arguments, and username. Values are replaced with '-', but for compatibility keys remain in place.

## 4.6 fileutils - Filesystem helpers

Virtually every Python programmer has used Python for wrangling disk contents, and fileutils collects solutions to some of the most commonly-found gaps in the standard library.

### 4.6.1 Creating, Finding, and Copying

Python’s os, os.path, and shutil modules provide good coverage of file wrangling fundamentals, and these functions help close a few remaining gaps.

boltons.fileutils.mkdir_p(path)

Creates a directory and any parent directories that may need to be created along the way, without raising errors for any existing directories. This function mimics the behavior of the mkdir -p command available in Linux/BSD environments, but also works on Windows.

boltons.fileutils.iter_find_files(directory, patterns, ignored=None, include_dirs=False)

Returns a generator that yields file paths under a directory, matching patterns using glob syntax (e.g., *.txt). Also supports ignored patterns.
Parameters

- **directory** *(str)* – Path that serves as the root of the search. Yielded paths will include this as a prefix.
- **patterns** *(str or list)* – A single pattern or list of glob-formatted patterns to find under *directory*.
- **ignored** *(str or list)* – A single pattern or list of glob-formatted patterns to ignore.
- **include_dirs** *(bool)* – Whether to include directories that match patterns, as well. Defaults to False.

For example, finding Python files in the current directory:

```python
>>> _CUR_DIR = os.path.dirname(os.path.abspath(__file__))
>>> filenames = sorted(iter_find_files(_CUR_DIR, '*.py'))
>>> os.path.basename(filenames[-1])
'urlutils.py'
```

Or, Python files while ignoring emacs lockfiles:

```python
>>> filenames = iter_find_files(_CUR_DIR, '*.py', ignored='.#*')
```

boltons.fileutils.copytree *(src, dst, symlinks=False, ignore=None)*

The `copytree()` function is an exact copy of the built-in `shutil.copytree()` function, with one key difference: it will not raise an exception if part of the tree already exists. It achieves this by using `mkdir_p()`.

Parameters

- **src** *(str)* – Path of the source directory to copy.
- **dst** *(str)* – Destination path. Existing directories accepted.
- **symlinks** *(bool)* – If True, copy symlinks rather than their contents.
- **ignore** *(callable)* – A callable that takes a path and directory listing, returning the files within the listing to be ignored.

For more details, check out `shutil.copytree()` and `shutil.copy2()`.

### 4.6.2 Atomic File Saving

Ideally, the road to success should never put current progress at risk. And that’s exactly why `atomic_save()` and `AtomicSaver` exist.

Using the same API as a writable file, all output is saved to a temporary file, and when the file is closed, the old file is replaced by the new file in a single system call, portable across all major operating systems. No more partially-written or partially-overwritten files.

boltons.fileutils.atomic_save *(dest_path, **kwargs)*

A convenient interface to the `AtomicSaver` type. See the `AtomicSaver` documentation for details.

class boltons.fileutils.AtomicSaver *(dest_path, **kwargs)*

`AtomicSaver` is a configurable context manager that provides a writable file which will be moved into place as long as no exceptions are raised within the context manager’s block. These “part files” are created in the same directory as the destination path to ensure atomic move operations (i.e., no cross-filesystem moves occur).

Parameters

- **dest_path** *(str)* – The path where the completed file will be written.
boltons Documentation, Release 20.1.0

- **overwrite** *(bool)* – Whether to overwrite the destination file if it exists at completion time. Defaults to True.

- **file_perms** *(int)* – Integer representation of file permissions for the newly-created file. Defaults are, when the destination path already exists, to copy the permissions from the previous file, or if the file did not exist, to respect the user’s configured `umask`, usually resulting in octal 0644 or 0664.

- **part_file** *(str)* – Name of the temporary part file. Defaults to `dest_path + .part`. Note that this argument is just the filename, and not the full path of the part file. To guarantee atomic saves, part files are always created in the same directory as the destination path.

- **overwrite_part** *(bool)* – Whether to overwrite the part_file, should it exist at setup time. Defaults to False, which results in an `OSError` being raised on pre-existing part files. Be careful of setting this to True in situations when multiple threads or processes could be writing to the same part file.

- **rm_part_on_exc** *(bool)* – Remove part_file on exception cases. Defaults to True, but False can be useful for recovery in some cases. Note that resumption is not automatic and by default an `OSError` is raised if the part_file exists.

Practically, the AtomicSaver serves a few purposes:

- Avoiding overwriting an existing, valid file with a partially written one.
- Providing a reasonable guarantee that a part file only has one writer at a time.
- Optional recovery of partial data in failure cases.

```python
boltons.fileutils.atomic_rename(src, dst, overwrite=False)
```

Rename `src` to `dst`, replacing `dst` if `overwrite` is True

```python
boltons.fileutils.replace(src, dst)
```

Similar to `os.replace()` in Python 3.3+, this function will atomically create or replace the file at path `dst` with the file at path `src`.

On Windows, this function uses the ReplaceFile API for maximum possible atomicity on a range of filesystems.

### 4.6.3 File Permissions

Linux, BSD, Mac OS, and other Unix-like operating systems all share a simple, foundational file permission structure that is commonly complicit in accidental access denial, as well as file leakage. `FilePerms` was built to increase clarity and cut down on permission-related accidents when working with files from Python code.

```python
class boltons.fileutils.FilePerms(user="", group="", other="")
```

The `FilePerms` type is used to represent standard POSIX filesystem permissions:

- Read
- Write
- Execute

Across three classes of user:

- Owning (u)ser
- Owner’s (g)roup
- Any (o)ther user

26 Chapter 4. Section listing
This class assists with computing new permissions, as well as working with numeric octal 777-style and rwx-style permissions. Currently it only considers the bottom 9 permission bits; it does not support sticky bits or more advanced permission systems.

Parameters

- **user** *(str)* – A string in the ‘rwx’ format, omitting characters for which owning user’s permissions are not provided.
- **group** *(str)* – A string in the ‘rwx’ format, omitting characters for which owning group permissions are not provided.
- **other** *(str)* – A string in the ‘rwx’ format, omitting characters for which owning other/world permissions are not provided.

There are many ways to use `FilePerms`:

```python
>>> FilePerms(user='rwx', group='xrw', other='wxr')  # note character order
FilePerms(user='rwx', group='xrw', other='xwx')
>>> int(FilePerms('r', 'r', ''))
288
>>> oct(288)[-3:]  # XXX Py3k
'440'
```

See also the `FilePerms.from_int()` and `FilePerms.from_path()` classmethods for useful alternative ways to construct `FilePerms` objects.

### 4.6.4 Miscellaneous

```python
class boltons.fileutils.DummyFile(path, mode='r', buffering=None)
```

#### 4.7 `formatutils - str.format()` toolbox

PEP 3101 introduced the `str.format()` method, and what would later be called “new-style” string formatting. For the sake of explicit correctness, it is probably best to refer to Python’s dual string formatting capabilities as **bracket-style** and **percent-style**. There is overlap, but one does not replace the other.

- Bracket-style is more pluggable, slower, and uses a method.
- Percent-style is simpler, faster, and uses an operator.

Bracket-style formatting brought with it a much more powerful toolbox, but it was far from a full one. `str.format()` uses more powerful syntax, but the tools and idioms for working with that syntax are not well-developed nor well-advertised.

`formatutils` adds several functions for working with bracket-style format strings:

- **DeferredValue**: Defer fetching or calculating a value until format time.
- **get_format_args()**: Parse the positional and keyword arguments out of a format string.
- **tokenize_format_str()**: Tokenize a format string into literals and `BaseFormatField` objects.
- **construct_format_field_str()**: Assists in programmatic construction of format strings.
- **infer_positional_format_args()**: Converts anonymous references in 2.7+ format strings to explicit positional arguments suitable for usage with Python 2.6.
class boltons.formatutils.DeferredValue(func, cache_value=True)

DeferredValue is a wrapper type, used to defer computing values which would otherwise be expensive to stringify and format. This is most valuable in areas like logging, where one would not want to waste time formatting a value for a log message which will subsequently be filtered because the message’s log level was DEBUG and the logger was set to only emit CRITICAL messages.

The :class: DeferredValue is initialized with a callable that takes no arguments and returns the value, which can be of any type. By default DeferredValue only calls that callable once, and future references will get a cached value. This behavior can be disabled by setting cache_value to False.

Parameters

- **func** *(function)* – A callable that takes no arguments and computes the value being represented.
- **cache_value** *(bool)* – Whether subsequent usages will call func again. Defaults to True.

```python
>>> import sys
>>> dv = DeferredValue(lambda: len(sys._current_frames()))
>>> output = "works great in all {0} threads!".format(dv)
```

PROTIP: To keep lines shorter, use: from formatutils import DeferredValue as DV

def get_value()  
    Computes, optionally caches, and returns the value of the func. If get_value() has been called before, a cached value may be returned depending on the cache_value option passed to the constructor.

boltons.formatutils.get_format_args(fstr)

Turn a format string into two lists of arguments referenced by the format string. One is positional arguments, and the other is named arguments. Each element of the list includes the name and the nominal type of the field.

```python
# >>> get_format_args("{noun} is {1:d} years old{punct}"))  
# (((1, <type 'int'>)), [('noun', <type 'str'>), ('punct', <type 'str'>)])
# XXX: Py3k >>> get_format_args("{noun} is {1:d} years old{punct}")) == (((1, int)), [('noun', str), ('punct', str)]) True
```

boltons.formatutils.tokenize_format_str(fstr, resolve_pos=True)

Takes a format string, turns it into a list of alternating string literals and BaseFormatField tokens. By default, also infers anonymous positional arguments into explicit, numbered positional references. To disable this behavior set resolve_pos to False.

boltons.formatutils.construct_format_field_str(fname, fspec, conv)

Constructs a format field string from the field name, spec, and conversion character (fname, fspec, conv). See Python String Formatting for more info.

boltons.formatutils.infer_positional_format_args(fstr)

Takes format strings with anonymous positional arguments, e.g., “[{}” and “{:d}”), and converts them into numbered ones for explicitness and compatibility with 2.6.

Returns a string with the inferred positional arguments.

class boltons.formatutils.BaseFormatField(fname, fspec=",", conv=None)

A class representing a reference to an argument inside of a bracket-style format string. For instance, in "{greeting}, world!", there is a field named “greeting”.

These fields can have many options applied to them. See the Python docs on Format String Syntax for the full details.

fstr

The current state of the field in string format.
There are only two built-in converters: `s` and `r`. They are somewhat rare and appearlike "(ref!r)".

**set_fname** (fname)
Set the field name.

**set_fspec** (fspec)
Set the field spec.

## 4.8 funcutils - functools fixes

Python’s built-in `functools` module builds several useful utilities on top of Python’s first-class function support. `funcutils` generally stays in the same vein, adding to and correcting Python’s standard metaprogramming facilities.

### Sections

- Decoration
- Function construction
- Improved partial
- Miscellaneous metaprogramming

### 4.8.1 Decoration

Decorators are among Python’s most elegant and succinct language features, and boltons adds one special function to make them even more powerful.

`boltons.funcutils.wraps (func, injected=None, expected=None, **kw)`
Decorator factory to apply `update_wrapper()` to a wrapper function.

Modeled after built-in `functools.wraps()`. Returns a decorator that invokes `update_wrapper()` with the decorated function as the wrapper argument and the arguments to `wraps()` as the remaining arguments. Default arguments are as for `update_wrapper()`. This is a convenience function to simplify applying `partial()` to `update_wrapper()`.

Same example as in `update_wrapper`’s doc but with `wraps`:

```python
>>> from boltons.funcutils import wraps

>>> def print_return(func):
    ...     @wraps(func)
    ...     def wrapper(*args, **kwargs):
    ...         ret = func(*args, **kwargs)
    ...         print(ret)
    ...         return ret
    ...     return wrapper
    ...

>>> @print_return
... def example():
...     '''docstring'''
...     return 'example return value'

>>> val = example()
```

(continues on next page)
4.8.2 Function construction

Functions are so key to programming in Python that there will even arise times where Python functions must be constructed in Python. Thankfully, Python is a dynamic enough to make this possible. Boltons makes it easy.

```python
例 example return value
>>> example.__name__
'example'
>>> example.__doc__
'docstring'
```

```python
class boltons.funcutils.FunctionBuilder (name, **kw)

The FunctionBuilder type provides an interface for programmatically creating new functions, either based on existing functions or from scratch.

Values are passed in at construction or set as attributes on the instance. For creating a new function based of an existing one, see the from_func() classmethod. At any point, get_func() can be called to get a newly compiled function, based on the values configured.

```python
>>> fb = FunctionBuilder('return_five', doc='returns the integer 5',
... body='return 5')
>>> f = fb.get_func()
>>> f()
5
>>> fb.varkw = 'kw'
>>> f_kw = fb.get_func()
>>> f_kw(ignored_arg='ignored_val')
5
```

Note that function signatures themselves changed quite a bit in Python 3, so several arguments are only applicable to FunctionBuilder in Python 3. Except for name, all arguments to the constructor are keyword arguments.

**Parameters**

- **name**(str) – Name of the function.
- **doc**(str) – Docstring for the function, defaults to empty.
- **module**(str) – Name of the module from which this function was imported. Defaults to None.
- **body**(str) – String version of the code representing the body of the function. Defaults to 'pass', which will result in a function which does nothing and returns None.
- **args**(list) – List of argument names, defaults to empty list, denoting no arguments.
- **varargs**(str) – Name of the catch-all variable for positional arguments. E.g., “args” if the resultant function is to have *args in the signature. Defaults to None.
- **varkw**(str) – Name of the catch-all variable for keyword arguments. E.g., “kwargs” if the resultant function is to have **kwargs in the signature. Defaults to None.
- **defaults**(tuple) – A tuple containing default argument values for those arguments that have defaults.
- **kwonlyargs**(list) – Argument names which are only valid as keyword arguments. Python 3 only.
• **kwonlydefaults** *(dict)* – A mapping, same as normal defaults, but only for the kwonlyargs. Python 3 only.

• **annotations** *(dict)* – Mapping of type hints and so forth. **Python 3 only.**

• **filename** *(str)* – The filename that will appear in tracebacks. Defaults to “boltons.funcutils.FunctionBuilder”.

• **indent** *(int)* – Number of spaces with which to indent the function body. Values less than 1 will result in an error.

• **dict** *(dict)* – Any other attributes which should be added to the functions compiled with this FunctionBuilder.

All of these arguments are also made available as attributes which can be mutated as necessary.

```python
add_arg(arg_name, default=NO_DEFAULT)
```

Add an argument with optional default (defaults to `funcutils.NO_DEFAULT`).

```python
classmethod from_func(func)
```

Create a new FunctionBuilder instance based on an existing function. The original function will not be stored or modified.

```python
get_defaults_dict()
```

Get a dictionary of function arguments with defaults and the respective values.

```python
get_func(execdict=None, add_source=True, with_dict=True)
```

Compile and return a new function based on the current values of the FunctionBuilder.

**Parameters**

- **execdict** *(dict)* – The dictionary representing the scope in which the compilation should take place. Defaults to an empty dict.

- **add_source** *(bool)* – Whether to add the source used to a special `__source__` attribute on the resulting function. Defaults to True.

- **with_dict** *(bool)* – Add any custom attributes, if applicable. Defaults to True.

To see an example of usage, see the implementation of `wraps()`.

```python
get_sig_str(with_annotations=True)
```

Return function signature as a string.

with_annotations is ignored on Python 2. On Python 3 signature will omit annotations if it is set to False.

```python
remove_arg(arg_name)
```

Remove an argument from this FunctionBuilder’s argument list. The resulting function will have one less argument per call to this function.

**Parameters**

- **arg_name** *(str)* – The name of the argument to remove.

  Raises a `ValueError` if the argument is not present.

### 4.8.3 Improved `partial`

`boltons.funcutils.partial` is an alias of `boltons.funcutils.CachedInstancePartial`.

```python
class boltons.funcutils.InstancePartial
```

`functools.partial` is a huge convenience for anyone working with Python’s great first-class functions. It allows developers to curry arguments and incrementally create simpler callables for a variety of use cases.
Unfortunately there’s one big gap in its usefulness: methods. Partials just don’t get bound as methods and automatically handed a reference to `self`. The `InstancePartial` type remedies this by inheriting from `functools.partial` and implementing the necessary descriptor protocol. There are no other differences in implementation or usage. `CachedInstancePartial`, below, has the same ability, but is slightly more efficient.

```python
class boltons.funcutils.CachedInstancePartial
```

The `CachedInstancePartial` is virtually the same as `InstancePartial`, adding support for method-usage to `functools.partial`, except that upon first access, it caches the bound method on the associated object, speeding it up for future accesses, and bringing the method call overhead to about the same as non-partial methods.

See the `InstancePartial` docstring for more details.

### 4.8.4 Miscellaneous metaprogramming

```python
boltons.funcutils.copy_function((orig, copy_dict=True))
```

Returns a shallow copy of the function, including code object, globals, closure, etc.

```python
>>> func = lambda: func
>>> func() is func
True
>>> func_copy = copy_function(func)
>>> func_copy() is func
True
>>> func_copy is not func
True
```

**Parameters**

- `orig (function)` – The function to be copied. Must be a function, not just any method or callable.
- `copy_dict (bool)` – Also copy any attributes set on the function instance. Defaults to `True`.

```python
boltons.funcutils.dir_dict(obj, raise_exc=False)
```

Return a dictionary of attribute names to values for a given object. Unlike `obj.__dict__`, this function returns all attributes on the object, including ones on parent classes.

```python
boltons.funcutils.mro_items(type_obj)
```

Takes a type and returns an iterator over all class variables throughout the type hierarchy (respecting the MRO).

```python
>>> sorted(set([k for k, v in mro_items(int) if not k.startswith('__') and 'bytes' not in k and not callable(v)]))
['denominator', 'imag', 'numerator', 'real']
```

```python
boltons.funcutils.format_invocation(name=', args=(), kwargs=None)
```

Given a name, positional arguments, and keyword arguments, format a basic Python-style function call.

```python
>>> print(format_invocation('func', args=(1, 2), kwargs={‘c’: 3}))
func(1, 2, c=3)
>>> print(format_invocation('a_func', args=(1,)))
a_func(1)
>>> print(format_invocation('kw_func', kwargs={‘a’: 1}, {‘b’, 2})))
kw_func(a=1, b=2)
```
boltons.funcutils.format_exp_repr(obj, pos_names, req_names=None, opt_names=None, opt_key=None)

Render an expression-style repr of an object, based on attribute names, which are assumed to line up with arguments to an initializer.

```python
>>> class Flag(object):
...     def __init__(self, length, width, depth=None):
...         self.length = length
...         self.width = width
...         self.depth = depth
...
That’s our Flag object, here are some example reprs for it:

```python
>>> flag = Flag(5, 10)
>>> print(format_exp_repr(flag, ['length', 'width'], [], ['depth']))
Flag(5, 10)
>>> flag2 = Flag(5, 15, 2)
>>> print(format_exp_repr(flag2, ['length'], ['width', 'depth']))
Flag(5, width=15, depth=2)
```

By picking the pos_names, req_names, opt_names, and opt_key, you can fine-tune how you want the repr to look.

**Parameters**

- **obj (object)** – The object whose type name will be used and attributes will be checked
- **pos_names (list)** – Required list of attribute names which will be rendered as positional arguments in the output repr.
- **req_names (list)** – List of attribute names which will always appear in the keyword arguments in the output repr. Defaults to None.
- **opt_names (list)** – List of attribute names which may appear in the keyword arguments in the output repr, provided they pass the opt_key check. Defaults to None.
- **opt_key (callable)** – A function or callable which checks whether an opt_name should be in the repr. Defaults to a None-check.

boltons.funcutils.format_nonexp_repr(obj, req_names=None, opt_names=None, opt_key=None)

Format a non-expression-style repr

Some object reprs look like object instantiation, e.g., App(r=[], mw=[]).

This makes sense for smaller, lower-level objects whose state roundtrips. But a lot of objects contain values that don’t roundtrip, like types and functions.

For those objects, there is the non-expression style repr, which mimic’s Python’s default style to make a repr look like so:

```python
>>> class Flag(object):
...     def __init__(self, length, width, depth=None):
...         self.length = length
...         self.width = width
...         self.depth = depth
...
>>> flag = Flag(5, 10)
>>> print(format_nonexp_repr(flag, ['length', 'width'], ['depth']))
<Flag length=5 width=10>
```
If no attributes are specified or set, utilizes the id, not unlike Python’s built-in behavior.

```python
>>> print(format_nonexp_repr(flag))
<Flag id=...>
```

## 4.9 gcutils - Garbage collecting tools

The Python Garbage Collector (GC) doesn’t usually get too much attention, probably because:

- Python’s reference counting effectively handles the vast majority of unused objects
- People are slowly learning to avoid implementing `object.__del__()`
- The collection itself strikes a good balance between simplicity and power (tunable generation sizes)
- The collector itself is fast and rarely the cause of long pauses associated with GC in other runtimes

Even so, for many applications, the time will come when the developer will need to track down:

- Circular references
- Misbehaving objects (locks, `__del__()`)
- Memory leaks
- Or just ways to shave off a couple percent of execution time

Thanks to the `gc` module, the GC is a well-instrumented entry point for exactly these tasks, and `gcutils` aims to facilitate it further.

`boltons.gcutils.get_all(type_obj, include_subtypes=True)`

Get a list containing all instances of a given type. This will work for the vast majority of types out there.

```python
>>> class Ratking(object): pass
>>> wiki, hak, sport = Ratking(), Ratking(), Ratking()
>>> len(get_all(Ratking))
3
```

However, there are some exceptions. For example, `get_all(bool)` returns an empty list because `True` and `False` are themselves built-in and not tracked.

```python
>>> get_all(bool)
[]
```

Still, it’s not hard to see how this functionality can be used to find all instances of a leaking type and track them down further using `gc.get_referrers()` and `gc.get_referents()`.

`get_all()` is optimized such that getting instances of user-created types is quite fast. Setting `include_subtypes` to `False` will further increase performance in cases where instances of subtypes aren’t required.

**Note:** There are no guarantees about the state of objects returned by `get_all()`, especially in concurrent environments. For instance, it is possible for an object to be in the middle of executing its `__init__()` and be only partially constructed.

```python
class boltons.gcutils.GCToggler(postcollect=False)
```

The GCToggler is a context-manager that allows one to safely take more control of your garbage collection schedule. Anecdotal experience says certain object-creation-heavy tasks see speedups of around 10% by simply doing one explicit collection at the very end, especially if most of the objects will stay resident.
Two GCTogglers are already present in the `gcutils` module:

- `toggle_gc` simply turns off GC at context entrance, and re-enables at exit
- `toggle_gc_postcollect` does the same, but triggers an explicit collection after re-enabling.

```python
>>> with toggle_gc:
...     x = [object() for i in range(1000)]
```

Between those two instances, the `GCToggler` type probably won’t be used much directly, but is documented for inheritance purposes.

```python
boltons.gcutils.toggle_gc = <boltons.gcutils.GCToggler object>
A context manager for disabling GC for a code block. See `GCToggler` for more details.

boltons.gcutils.toggle_gc_postcollect = <boltons.gcutils.GCToggler object>
A context manager for disabling GC for a code block, and collecting before re-enabling. See `GCToggler` for more details.
```

### 4.10 `ioutils` - Input/output enhancements

Module `ioutils` implements a number of helper classes and functions which are useful when dealing with input, output, and bytestreams in a variety of ways.

#### 4.10.1 Spooled Temporary Files

Spooled Temporary Files are file-like objects that start out mapped to in-memory objects, but automatically roll over to a temporary file once they reach a certain (configurable) threshold. Unfortunately the built-in SpooledTemporaryFile class in Python does not implement the exact API that some common classes like StringIO do. SpooledTemporaryFile also spools all of it’s in-memory files as cStringIO instances. cStringIO instances cannot be deep-copied, and they don’t work with the zip library either. This along with the incompatible api makes it useless for several use-cases.

To combat this but still gain the memory savings and usefulness of a true spooled file-like-object, two custom classes have been implemented which have a compatible API.

**SpooledBytesIO**

```python
class boltons.ioutils.SpooledBytesIO(max_size=5000000, dir=None)
SpooledBytesIO is a spooled file-like-object that only accepts bytes. On Python 2.x this means the ‘str’ type; on Python 3.x this means the ‘bytes’ type. Bytes are written in and retrieved exactly as given, but it will raise TypeErrors if something other than bytes are written.
```

Example:

```python
>>> from boltons import ioutils
>>> with ioutils.SpooledBytesIO() as f:
...     f.write(b"Happy IO")
...     _ = f.seek(0)
...     isinstance(f.getvalue(), ioutils.binary_type)
True
```
SpooledStringIO

class boltons.ioutils.SpooledStringIO(*args, **kwargs)
SpooledStringIO is a spooled file-like-object that only accepts unicode values. On Python 2.x this means the 'unicode' type and on Python 3.x this means the 'str' type. Values are accepted as unicode and then coerced into utf-8 encoded bytes for storage. On retrieval, the values are returned as unicode.

Example:

```python
>>> from boltons import ioutils
>>> with ioutils.SpooledStringIO() as f:
...     f.write(u"\u2014 Hey, an emdash!")
...     _ = f.seek(0)
...     isinstance(f.read(), ioutils.text_type)
True
```

4.10.2 Examples

It’s not uncommon to find excessive usage of StringIO in older Python code. A SpooledTemporaryFile would be a nice replacement if one wanted to reduce memory overhead, but unfortunately its api differs too much. This is a good candidate for SpooledBytesIO as it is api compatible and thus may be used as a drop-in replacement.

Old Code:

```python
flo = StringIO()
flo.write(gigantic_string)
```

Updated:

```python
from boltons.ioutils import SpooledBytesIO
flo = SpooledBytesIO()
flo.write(gigantic_string)
```

Another good use case is downloading a file from some remote location. It’s nice to keep it in memory if it’s small, but writing a large file into memory can make servers quite grumpy. If the file being downloaded happens to be a zip file then things are worse. You can’t use a normal SpooledTemporaryFile because it isn’t compatible. A SpooledBytesIO instance is a good alternative. Here is a simple example using the requests library to download a zip file:

```python
from zipfile import ZipFile
import requests
from boltons import ioutils

# Using a context manager with stream=True ensures the connection is closed. See:
# http://docs.python-requests.org/en/master/user/advanced/#body-content-workflow
with requests.get("http://127.0.0.1/test_file.zip", stream=True) as r:
    if r.status_code == 200:
        with ioutils.SpooledBytesIO() as flo:
            for chunk in r.iter_content(chunk_size=64000):
                flo.write(chunk)

        flo.seek(0)
        zip_doc = ZipFile(flo)
```

(continues on next page)
# Print all the files in the zip
print(zip_doc.namelist())

## 4.10.3 Multiple Files

### MultiFileReader

```python
class boltons.ioutils.MultiFileReader(*fileobjs):
    Takes a list of open files or file-like objects and provides an interface to read from them all contiguously. Like itertools.chain(), but for reading files.

>>> mfr = MultiFileReader(BytesIO(b'ab'), BytesIO(b'cd'), BytesIO(b'e'))
>>> mfr.read(3).decode('ascii')
'u'abc'
>>> mfr.read(3).decode('ascii')
'u'de'
```

The constructor takes as many fileobjs as you hand it, and will raise a TypeError on non-file-like objects. A ValueError is raised when file-like objects are a mix of bytes- and text-handling objects (for instance, BytesIO and StringIO).

## 4.11 `iterutils` - `itertools` improvements

`itertools` is full of great examples of Python generator usage. However, there are still some critical gaps. `iterutils` fills many of those gaps with featureful, tested, and Pythonic solutions.

Many of the functions below have two versions, one which returns an iterator (denoted by the `*_iter` naming pattern), and a shorter-named convenience form that returns a list. Some of the following are based on examples in `itertools` docs.

### Sections

- **Iteration**
- **Nested**
- **Numeric**
- **Categorization**
- **Sorting**
- **Reduction**
- **Type Checks**

### 4.11.1 Iteration

These are generators and convenient list-producing counterparts comprising several common patterns of iteration not present in the standard library.
boltons.iterutils.split(src, sep=None, maxsplit=None)

Splits an iterable based on a separator. Like str.split(), but for all iterables. Returns a list of lists.

```python
tests = ['hi', 'hello', None, None, 'sup', None, 'soap', None]
print(boltons.iterutils.split(tests))
```

```
[['hi', 'hello'], ['sup'], ['soap']]
```

See split_iter() docs for more info.

boltons.iterutils.split_iter(src, sep=None, maxsplit=None)

Splits an iterable based on a separator, sep, a max of maxsplit times (no max by default). sep can be:

- a single value
- an iterable of separators
- a single-argument callable that returns True when a separator is encountered

split_iter() yields lists of non-separator values. A separator will never appear in the output.

```python
tests = ['hi', 'hello', None, None, 'sup', None, 'soap', None]
print(list(boltons.iterutils.split_iter(tests)))
```

```
[['hi', 'hello'], [], ['sup'], []]
```

Note that split_iter is based on str.split(), so if sep is None, split() groups separators. If empty lists are desired between two contiguous None values, simply use sep=[None]:

```python
tests = ['hi', 'hello', None, None, 'sup', None, 'soap', None]
print(list(boltons.iterutils.split_iter(tests, sep=[None])))
```

```
[['hi', 'hello'], [], ['sup'], [], ['soap'], []]
```

Using a callable separator:

```python
def falsy_sep(x):
    return not x
tests = ['hi', 'hello', None, '', 'sup', False]
print(list(boltons.iterutils.split_iter(tests, falsy_sep)))
```

```
[['hi', 'hello'], [], ['sup'], [], ['soap'], []]
```

See split() for a list-returning version.

boltons.iterutils.chunked(src, size, count=None, **kw)

Returns a list of count chunks, each with size elements, generated from iterable src. If src is not evenly divisible by size, the final chunk will have fewer than size elements. Provide the fill keyword argument to provide a pad value and enable padding, otherwise no padding will take place.

```python
tests = list(range(10))
print(boltons.iterutils.chunked(tests, 3))
```

```
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]
```

See chunked_iter() for more info.

boltons.iterutils.chunked_iter(src, size, **kw)

Generates size-sized chunks from src iterable. Unless the optional fill keyword argument is provided, iterables not evenly divisible by size will have a final chunk that is smaller than size.

```python
tests = list(range(10))
print(list(boltons.iterutils.chunked_iter(tests, 3)))
```

```
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]
```

```python
tests = list(range(10))
print(list(boltons.iterutils.chunked_iter(tests, 3, fill=None)))
```

```
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, None, None]]
```
Note that `fill=None` in fact uses `None` as the fill value.

boltons.iterutils.pairwise(src)
Convenience function for calling `windowed()` on `src`, with `size` set to 2.

```python
>>> pairwise(range(5))
[(0, 1), (1, 2), (2, 3), (3, 4)]
>>> pairwise([])
[]
```

The number of pairs is always one less than the number of elements in the iterable passed in, except on empty inputs, which returns an empty list.

boltons.iterutils.pairwise_iter(src)
Convenience function for calling `windowed_iter()` on `src`, with `size` set to 2.

```python
>>> list(pairwise_iter(range(5)))
[(0, 1), (1, 2), (2, 3), (3, 4)]
>>> list(pairwise_iter([]))
[]
```

The number of pairs is always one less than the number of elements in the iterable passed in, or zero, when `src` is empty.

boltons.iterutils.windowed(src, size)
Returns tuples with exactly length `size`. If the iterable is too short to make a window of length `size`, no tuples are returned. See `windowed_iter()` for more.

boltons.iterutils.windowed_iter(src, size)
Returns tuples with length `size` which represent a sliding window over iterable `src`.

```python
>>> list(windowed_iter(range(7), 3))
[(0, 1, 2), (1, 2, 3), (2, 3, 4), (3, 4, 5), (4, 5, 6)]
```

If the iterable is too short to make a window of length `size`, then no window tuples are returned.

```python
>>> list(windowed_iter(range(3), 5))
[]
```

boltons.iterutils.unique(src, key=None)
`unique()` returns a list of unique values, as determined by `key`, in the order they first appeared in the input iterable, `src`.

```python
>>> ones_n_zeros = '11010110001010010101010'
>>> ''.join(unique(ones_n_zeros))
'10'
```

See `unique_iter()` docs for more details.

boltons.iterutils.unique_iter(src, key=None)
Yield unique elements from the iterable, `src`, based on `key`, in the order in which they first appeared in `src`.

```python
>>> repetitious = [1, 2, 3] * 10
>>> list(unique_iter(repetitious))
[1, 2, 3]
```

By default, `key` is the object itself, but `key` can either be a callable or, for convenience, a string name of the attribute on which to uniqueify objects, falling back on identity when the attribute is not present.

4.11. iterutils - itertools improvements
4.11.2 Nested

Nested data structures are common. Yet virtually all of Python’s compact iteration tools work with flat data: list comprehensions, map/filter, generator expressions, itertools, even other iterutils.

The functions below make working with nested iterables and other containers as succinct and powerful as Python itself.

The remap (“recursive map”) function is used to traverse and transform nested structures. Lists, tuples, sets, and dictionaries are just a few of the data structures nested into heterogenous tree-like structures that are so common in programming. Unfortunately, Python’s built-in ways to manipulate collections are almost all flat. List comprehensions may be fast and succinct, but they do not recurse, making it tedious to apply quick changes or complex transforms to real-world data.

remap goes where list comprehensions cannot.

Here’s an example of removing all Nones from some data:

```python
comprehensions = ['hi', 'hello', 'ok', 'bye', 'yes']
print(list(unique_iter(comprehensions, key=lambda x: len(x))))

['hi', 'hello', 'bye']

remap(root, visit=<function default_visit>, enter=<function default_enter>, exit=<function default_exit>, **kwargs)
```

Parameters

- **root** – The target object to traverse. By default, remap supports iterables like list, tuple, dict, and set, but any object traversable by enter will work.

- **visit (callable)** – This function is called on every item in root. It must accept three positional arguments, path, key, and value. path is simply a tuple of parents’ keys. visit should return the new key-value pair. It may also return True as shorthand to keep the old item unmodified, or False to drop the item from the new structure. visit is called after enter, on the new parent.

  The visit function is called for every item in root, including duplicate items. For traversable values, it is called on the new parent object, after all its children have been visited. The default visit behavior simply returns the key-value pair unmodified.

- **enter (callable)** – This function controls which items in root are traversed. It accepts the same arguments as visit: the path, the key, and the value of the current item. It returns a pair of the blank new parent, and an iterator over the items which should be visited. If False is returned instead of an iterator, the value will not be traversed.

  The enter function is only called once per unique value. The default enter behavior support mappings, sequences, and sets. Strings and all other iterables will not be traversed.
• **exit (callable)** – This function determines how to handle items once they have been visited. It gets the same three arguments as the other functions – *path*, *key*, *value* – plus two more: the blank new parent object returned from *enter*, and a list of the new items, as remapped by *visit*.

Like *enter*, the *exit* function is only called once per unique value. The default exit behavior is to simply add all new items to the new parent, e.g., using `list.extend()` and `dict.update()` to add to the new parent. Immutable objects, such as a tuple or `namedtuple`, must be recreated from scratch, but use the same type as the new parent passed back from the *enter* function.

• **reraise_visit (bool)** – A pragmatic convenience for the *visit* callable. When set to `False`, remap ignores any errors raised by the *visit* callback. Items causing exceptions are kept. See examples for more details.

remap is designed to cover the majority of cases with just the *visit* callable. While passing in multiple callables is very empowering, remap is designed so very few cases should require passing more than one function.

When passing *enter* and *exit*, it’s common and easiest to build on the default behavior. Simply add `from boltons.iterutils import default_enter` (or `default_exit`), and have your enter/exit function call the default behavior before or after your custom logic. See this example.

Duplicate and self-referential objects (aka reference loops) are automatically handled internally, as shown here.

```python
boltons.iterutils.get_path(root, path=('a', 'b', 'c', 2, 0))
```

Retrieve a value from a nested object via a tuple representing the lookup path.

>>> root = {'a': {'b': {'c': [[1], [2], [3]]}}}

>>> get_path(root, ('a', 'b', 'c', 2, 0))
3

The path format is intentionally consistent with that of `remap()`.

One of get_path’s chief aims is improved error messaging. EAFP is great, but the error messages are not.

For instance, `root['a']['b']['c'][2][1]` gives back `IndexError: list index out of range`

What went out of range where? get_path currently raises `PathAccessError: could not access 2 from path ('a', 'b', 'c', 2, 1), got error: IndexError('list index out of range',), a subclass of IndexError and KeyError.

You can also pass a default that covers the entire operation, should the lookup fail at any level.

**Parameters**

- **root** – The target nesting of dictionaries, lists, or other objects supporting __getitem__.
- **path (tuple)** – A list of strings and integers to be successively looked up within *root*.
- **default** – The value to be returned should any PathAccessError exceptions be raised.

```python
boltons.iterutils.research(root, query=<function <lambda>>, reraise=False)
```

The `research()` function uses `remap()` to recurse over any data nested in *root*, and find values which match a given criterion, specified by the *query* callable.

Results are returned as a list of (path, value) pairs. The paths are tuples in the same format accepted by `get_path()`. This can be useful for comparing values nested in two or more different structures.

Here’s a simple example that finds all integers:
Note how `query` follows the same, familiar `path`, `key`, `value` signature as the `visit` and `enter` functions on `remap()`, and returns a `bool`.

**Parameters**

- `root` – The target object to search. Supports the same types of objects as `remap()`, including `list`, `tuple`, `dict`, and `set`.

- `query` (`callable`) – The function called on every object to determine whether to include it in the search results. The callable must accept three arguments, `path`, `key`, and `value`, commonly abbreviated `p`, `k`, and `v`, same as `enter` and `visit` from `remap()`.

- `reraise` ( `bool`) – Whether to reraise exceptions raised by `query` or to simply drop the result that caused the error.

With `research()` it’s easy to inspect the details of a data structure, like finding values that are at a certain depth (using `len(p)`) and much more. If more advanced functionality is needed, check out the code and make your own `remap()` wrapper, and consider submitting a patch!

### `boltons.iterutils.flatten` ( `iterable`)

`flatten()` returns a collapsed list of all the elements from `iterable` while collapsing any nested iterables.

```python
>>> nested = [[[1, 2], [[3], [4, 5]]]
>>> flatten(nested)
[1, 2, 3, 4, 5]
```

### `boltons.iterutils.flatten_iter` ( `iterable`)

`flatten_iter()` yields all the elements from `iterable` while collapsing any nested iterables.

```python
>>> nested = [[[1, 2], [[3], [4, 5]]]
>>> list(flatten_iter(nested))
[1, 2, 3, 4, 5]
```

### 4.11.3 Numeric

Number sequences are an obvious target of Python iteration, such as the built-in `range()`, `xrange()`, and `itertools.count()`. Like the `Iteration` members above, these return iterators and lists, but take numeric inputs instead of iterables.

### `boltons.iterutils.backoff` ( `start` , `stop`, `count=None`, `factor=2.0`, `jitter=False`)

Returns a list of geometrically-increasing floating-point numbers, suitable for usage with exponential backoff. Exactly like `backoff_iter()`, but without the 'repeat' option for `count`. See `backoff_iter()` for more details.

```python
>>> backoff(1, 10)
[1.0, 2.0, 4.0, 8.0, 10.0]
```

### `boltons.iterutils.backoff_iter` ( `start` , `stop`, `count=None`, `factor=2.0`, `jitter=False`)

Generates a sequence of geometrically-increasing floats, suitable for usage with exponential backoff. Starts with `start`, increasing by `factor` until `stop` is reached, optionally stopping iteration once `count` numbers are yielded. `factor` defaults to 2. In general retrying with properly-configured backoff creates a better-behaved component for a larger service ecosystem.
>>> list(backoff_iter(1.0, 10.0, count=5))
[1.0, 2.0, 4.0, 8.0, 10.0]
>>> list(backoff_iter(1.0, 10.0, count=8))
[1.0, 2.0, 4.0, 8.0, 10.0, 10.0, 10.0, 10.0]
>>> list(backoff_iter(0.25, 100.0, factor=10))
[0.25, 2.5, 25.0, 100.0]

A simplified usage example:

```python
for timeout in backoff_iter(0.25, 5.0):
    try:
        res = network_call()
        break
    except Exception as e:
        log(e)
        time.sleep(timeout)
```

An enhancement for large-scale systems would be to add variation, or jitter, to timeout values. This is done to avoid a thundering herd on the receiving end of the network call.

Finally, for count, the special value 'repeat' can be passed to continue yielding indefinitely.

**Parameters**

- **start** (*float*) – Positive number for baseline.
- **stop** (*float*) – Positive number for maximum.
- **count** (*int*) – Number of steps before stopping iteration. Defaults to the number of steps between start and stop. Pass the string, 'repeat', to continue iteration indefinitely.
- **factor** (*float*) – Rate of exponential increase. Defaults to 2.0, e.g., [1, 2, 4, 8, 16].
- **jitter** (*float*) – A factor between -1.0 and 1.0, used to uniformly randomize and thus spread out timeouts in a distributed system, avoiding rhythm effects. Positive values use the base backoff curve as a maximum, negative values use the curve as a minimum. Set to 1.0 or True for a jitter approximating Ethernet’s time-tested backoff solution. Defaults to False.

**boltons.iterutils.frange** *(stop, start=None, step=1.0)*

A range() clone for float-based ranges.

```python
>>> frange(5)
[0.0, 1.0, 2.0, 3.0, 4.0]
>>> frange(6, step=1.25)
[0.0, 1.25, 2.5, 3.75, 5.0]
>>> frange(100.5, 101.5, 0.25)
[100.5, 100.75, 101.0, 101.25]
>>> frange(5, 0)
[]
>>> frange(5, 0, step=-1.25)
[5.0, 3.75, 2.5, 1.25]
```

**boltons.iterutils.xfrange** *(stop, start=None, step=1.0)*

Same as frange(), but generator-based instead of returning a list.

```python
>>> tuple(xfrange(1, 3, step=0.75))
(1.0, 1.75, 2.5)
```

See frange() for more details.
4.11.4 Categorization

These functions operate on iterables, dividing into groups based on a given condition.

`boltons.iterutils.bucketize(src, key=<type 'bool'>, value_transform=None, key_filter=None)`

Group values in the `src` iterable by the value returned by `key`.

```python
>>> bucketize(range(5))
{False: [0], True: [1, 2, 3, 4])
>>> is_odd = lambda x: x % 2 == 1
>>> bucketize(range(5), is_odd)
{False: [0, 2, 4], True: [1, 3])
```

`key` is `bool` by default, but can either be a callable or a string name of the attribute on which to bucketize objects.

```python
>>> bucketize([1+1j, 2+2j, 1, 2], key='real')
{1.0: [(1+1j), 1], 2.0: [(2+2j), 2])
```

Value lists are not deduplicated:

```python
>>> bucketize([None, None, None, 'hello'])
{False: [None, None, None], True: ['hello'])
```

Bucketize into more than three groups

```python
>>> bucketize(range(10), lambda x: x % 3)
{0: [0, 3, 6, 9], 1: [1, 4, 7], 2: [2, 5, 8])
```

`bucketize` has a couple of advanced options useful in certain cases. `value_transform` can be used to modify values as they are added to buckets, and `key_filter` will allow excluding certain buckets from being collected.

```python
>>> bucketize(range(5), value_transform=lambda x: x*x)
{0: [0], True: [1, 4, 9, 16])
```

```python
>>> bucketize(range(10), key=lambda x: x % 3, key_filter=lambda k: k % 3 != 1)
{0: [0, 3, 6, 9], 2: [2, 5, 8])
```

Note in some of these examples there were at most two keys, `True` and `False`, and each key present has a list with at least one item. See `partition()` for a version specialized for binary use cases.

`boltons.iterutils.partition(src, key=<type 'bool'>)`

No relation to `str.partition()`, `partition` is like `bucketize()`, but for added convenience returns a tuple of `(truthy_values, falsy_values)`.

```python
>>> nonempty, empty = partition(["", ", hi", ", bye")
>>> nonempty
['hi", 'bye")
```

`key` defaults to `bool`, but can be carefully overridden to use either a function that returns either `True` or `False` or a string name of the attribute on which to partition objects.

```python
>>> import string
>>> is_digit = lambda x: x in string.digits
>>> decimal_digits, hexletters = partition(string.hexdigits, is_digit)
>>> ".join(decimal_digits), ".join(hexletters)
('0123456789", 'abcdefABCDEF')
```
4.11.5 Sorting

The built-in `sorted()` is great, but what do you do when you want to partially override the sort order?

`boltons.iterutils.soft_sorted(iterable, first=None, last=None, key=None, reverse=False)`

For when you care about the order of some elements, but not about others.

Use this to float to the top and/or sink to the bottom a specific ordering, while sorting the rest of the elements according to normal `sorted()` rules.

```python
>>> soft_sorted(['two', 'b', 'one', 'a'], first=['one', 'two'])
['one', 'two', 'a', 'b']
>>> soft_sorted(range(7), first=[6, 15], last=[2, 4], reverse=True)
[6, 5, 3, 1, 0, 2, 4]
>>> import string
>>> ''.join(soft_sorted(string.hexdigits, first='za1', last='b', key=str.lower))
'aA1023456789cCdDeEfFbB'
```

Parameters

- `iterable (list)` – A list or other iterable to sort.
- `first (list)` – A sequence to enforce for elements which should appear at the beginning of the returned list.
- `last (list)` – A sequence to enforce for elements which should appear at the end of the returned list.
- `key (callable)` – Callable used to generate a comparable key for each item to be sorted, same as the key in `sorted()`. Note that entries in `first` and `last` should be the keys for the items. Defaults to `passthrough`/the identity function.
- `reverse (bool)` – Whether or not elements not explicitly ordered by `first` and `last` should be in reverse order or not.

Returns a new list in sorted order.

4.11.6 Reduction

`reduce()` is a powerful function, but it is also very open-ended and not always the most readable. The standard library recognized this with the addition of `sum()`, `all()`, and `any()`. All these functions take a basic operator (+, and, and or) and use the operator to turn an iterable into a single value.

Functions in this category follow that same spirit, turning iterables like lists into single values:

`boltons.iterutils.one (src, default=None, key=None)`

Along the same lines as builtins, `all()` and `any()`, and similar to `first()`, `one()` returns the single object in the given iterable `src` that evaluates to `True`, as determined by callable `key`. If unset, `key` defaults to `bool`. If no such objects are found, `default` is returned. If `default` is not passed, `None` is returned.

If `src` has more than one object that evaluates to `True`, or if there is no object that fulfills such condition, return `default`. It’s like an XOR over an iterable.

```python
>>> one((True, False, False))
True
>>> one((True, False, True))
>>> one((0, 0, 'a'))
'a'
```

(continues on next page)
>>> one((0, False, None))
False
>>> one((True, True), default=False)
False
>>> bool(one(('', 1)))
True
>>> one((10, 20, 30, 42), key=lambda i: i > 40)
42

See Martín Gaitán’s original repo for further use cases.

boltons.iterutils.first (iterable, default=None, key=None)
Return first element of iterable that evaluates to True, else return None or optional default. Similar to one().

>>> first([0, False, None, [], (), 42])
42
>>> first([0, False, None, [], ()]) is None
True
>>> first([0, False, None, [], ()], default='ohai')
'ohai'
>>> import re
>>> m = first(re.match(regex, 'abc') for regex in ['b.*', 'a(.*)'])
>>> m.group(1)
'bc'

The optional key argument specifies a one-argument predicate function like that used for filter(). The key argument, if supplied, should be in keyword form. For example, finding the first even number in an iterable:

>>> first([1, 1, 3, 4, 5], key=lambda x: x % 2 == 0)
4

Contributed by Hynek Schlawack, author of the original standalone module.

boltons.iterutils.same (iterable, ref=Sentinel('_UNSET'))
same() returns True when all values in iterable are equal to one another, or optionally a reference value, ref. Similar to all() and any() in that it evaluates an iterable and returns a bool. same() returns True for empty iterables.

>>> same([])
True
>>> same([1])
True
>>> same(['a', 'a', 'a'])
True
>>> same(range(20))
False
>>> same([[]])
True
>>> same([[]], ref='test')
False

4.11.7 Type Checks

In the same vein as the feature-checking built-in, callable().

boltons.iterutils.is_iterable (obj)
Similar in nature to callable(), is_iterable returns True if an object is iterable, False if not.
boltons.iterutils.is_iterable([])
True
>>> is_iterable(object())
False

boltons.iterutils.is_scalar(obj)
A near-mirror of is_iterable(). Returns False if an object is an iterable container type. Strings are considered scalar as well, because strings are more often treated as whole values as opposed to iterables of 1-character substrings.

>>> is_scalar(object())
True
>>> is_scalar(range(10))
False
>>> is_scalar('hello')
True

boltons.iterutils.is_collection(obj)
The opposite of is_scalar(). Returns True if an object is an iterable other than a string.

>>> is_collection(object())
False
>>> is_collection(range(10))
True
>>> is_collection('hello')
False

4.12 jsonutils - JSON interactions

jsonutils aims to provide various helpers for working with JSON. Currently it focuses on providing a reliable and intuitive means of working with JSON Lines-formatted files.

```python
class boltons.jsonutils.JSONLIterator(file_obj, ignore_errors=False, reverse=False, rel_seek=None)
```
The JSONLIterator is used to iterate over JSON-encoded objects stored in the JSON Lines format (one object per line).

Most notably it has the ability to efficiently read from the bottom of files, making it very effective for reading in simple append-only JSONL use cases. It also has the ability to start from anywhere in the file and ignore corrupted lines.

**Parameters**

- `file_obj (file)` – An open file object.
- `ignore_errors (bool)` – Whether to skip over lines that raise an error on deserialization (json.loads()).
- `reverse (bool)` – Controls the direction of the iteration. Defaults to False. If set to True and `rel_seek` is unset, seeks to the end of the file before iteration begins.
- `rel_seek (float)` – Used to preseek the start position of iteration. Set to 0.0 for the start of the file, 1.0 for the end, and anything in between.

**cur_byte_pos**
A property representing where in the file the iterator is reading.
next()
Yields one dict loaded with json.loads(), advancing the file object by one line. Raises
StopIteration upon reaching the end of the file (or beginning, if reverse was set to True.

boltons$jsonutils$.reverse_iter_lines(file_obj, blocksize=4096, preseek=True)
Returns an iterator over the lines from a file object, in reverse order, i.e., last line first, first line last. Uses the
file.seek() method of file objects, and is tested compatible with file objects, as well as StringIO.

Parameters
- file_obj(file) – An open file object. Note that reverse_iter_lines mutably
  reads from the file and other functions should not mutably interact with the file object.
- blocksize(int) – The block size to pass to file.read()
- preseek(bool) – Tells the function whether or not to automatically seek to the end of the
  file. Defaults to True. preseek=False is useful in cases when the file cursor is already
  in position, either at the end of the file or in the middle for relative reverse line generation.

4.13 listutils - list derivatives

Python’s builtin list is a very fast and efficient sequence type, but it could be better for certain access patterns, such
as non-sequential insertion into a large lists. listutils provides a pure-Python solution to this problem.

For utilities for working with iterables and lists, check out iterutils. For the a list-based version of
collections.namedtuple, check out namedutils.

boltons.listutils.BList
alias of boltons.listutils.BarrelList

class boltons.listutils.BarrelList(iterable=None)
The BarrelList is a list subtype backed by many dynamically-scaled sublists, to provide better scaling
and random insertion/deletion characteristics. It is a subtype of the builtin list and has an identical API,
supporting indexing, slicing, sorting, etc. If application requirements call for something more performant,
consider the blist module available on PyPI.

The name comes by way of Kurt Rose, who said it reminded him of barrel shifters. Not sure how, but it’s
BList-like, so the name stuck. BList is of course a reference to B-trees.

Parameters iterable – An optional iterable of initial values for the list.

```python
>>> blist = BList(xrange(100000))
>>> blist.pop(50000)
50000
>>> len(blist)
99999
>>> len(blist.lists)  # how many underlying lists
8
>>> slice_idx = blist.lists[0][-1]
>>> blist[slice_idx:slice_idx + 2]
BarrelList([11637, 11638])
```

Slicing is supported and works just fine across list borders, returning another instance of the BarrelList.

append(item)
L.append(object) – append object to end

count(value) → integer – return number of occurrences of value
del_slice(start, stop, step=None)

extend(iterable)
   L.extend(iterable) – extend list by appending elements from the iterable

classmethod from_iterable(it)

index(value[, start[, stop]]) → integer – return first index of value.
   Raises ValueError if the value is not present.

insert(index, item)
   L.insert(index, object) – insert object before index

iter_slice(start, stop, step=None)

pop([index]) → item – remove and return item at index (default last).
   Raises IndexError if list is empty or index is out of range.

reverse()
   L.reverse() – reverse IN PLACE

sort()
   L.sort(cmp=None, key=None, reverse=False) – stable sort IN PLACE; cmp(x, y) -> -1, 0, 1

4.14 mathutils - Mathematical functions

This module provides useful math functions on top of Python's built-in math module.

class boltons.mathutils.Bits(val=0, len_=None)
   An immutable bit-string or bit-array object. Provides list-like access to bits as bools, as well as bitwise masking
   and shifting operators. Bits also make it easy to convert between many different useful representations:
   • bytes – good for serializing raw binary data
   • int – good for incrementing (e.g. to try all possible values)
   • list of bools – good for iterating over or treating as flags
   • hex/bin string – good for human readability

as_bin()

as_bytes()

as_hex()

as_int()

as_list()

classmethod from_bin(bin)

classmethod from_bytes(bytes_)

classmethod from_hex(hex)

classmethod from_int(int_, len_=None)

classmethod from_list(list_)

len

val
4.14.1 Alternative Rounding Functions

`boltons.mathutils.clamp(x, lower=-inf, upper=inf)`

Limit a value to a given range.

**Parameters**

- `x (int or float)` – Number to be clamped.
- `lower (int or float)` – Minimum value for x.
- `upper (int or float)` – Maximum value for x.

The returned value is guaranteed to be between `lower` and `upper`. Integers, floats, and other comparable types can be mixed.

```python
>>> clamp(1.0, 0, 5)
1.0
>>> clamp(-1.0, 0, 5)
0
>>> clamp(101.0, 0, 5)
5
>>> clamp(123, upper=5)
5
```

Similar to numpy’s `clip` function.

`boltons.mathutils.ceil(x, options=None)`

Return the ceiling of `x`. If `options` is set, return the smallest integer or float from `options` that is greater than or equal to `x`.

**Parameters**

- `x (int or float)` – Number to be tested.
- `options (iterable)` – Optional iterable of arbitrary numbers (ints or floats).

```python
>>> VALID_CABLE_CSA = [1.5, 2.5, 4, 6, 10, 25, 35, 50]
>>> ceil(3.5, options=VALID_CABLE_CSA)
4
>>> ceil(2.5, options=VALID_CABLE_CSA)
2.5
```

`boltons.mathutils.floor(x, options=None)`

Return the floor of `x`. If `options` is set, return the largest integer or float from `options` that is less than or equal to `x`.

**Parameters**

- `x (int or float)` – Number to be tested.
- `options (iterable)` – Optional iterable of arbitrary numbers (ints or floats).

```python
>>> VALID_CABLE_CSA = [1.5, 2.5, 4, 6, 10, 25, 35, 50]
>>> floor(3.5, options=VALID_CABLE_CSA)
3
>>> floor(2.5, options=VALID_CABLE_CSA)
2
```

Note: `ceil()` and `floor()` functions are based on this example using from the `bisect` module in the standard library. Refer to this StackOverflow Answer for further information regarding the performance impact of this approach.
4.15 mboxutils - Unix mailbox utilities

Useful utilities for working with the mbox-formatted mailboxes. Credit to Mark Williams for these.

```python
class boltons.mboxutils.mbox_readonlydir(path, factory=None, create=True, maxmem=1048576)
```

A subclass of `mailbox.mbox` suitable for use with mboxes inside a read-only mail directory, e.g., `/var/mail`. Otherwise the API is exactly the same as the built-in `mbox`.

Deletes messages via truncation, in the manner of Heirloom mailx.

**Parameters**

- `path` (*str*) – Path to the mbox file.
- `factory` (*type*) – Message type (defaults to `rfc822.Message`)
- `create` (*bool*) – Create mailbox if it does not exist. (defaults to `True`)
- `maxmem` (*int*) – Specifies, in bytes, the largest sized mailbox to attempt to copy into memory. Larger mailboxes will be copied incrementally which is more hazardous. (defaults to 4MB)

**Note:** Because this truncates and rewrites parts of the mbox file, this class can corrupt your mailbox. Only use this if you know the built-in `mailbox.mbox` does not work for your use case.

**flush()**

Write any pending changes to disk. This is called on mailbox close and is usually not called explicitly.

**Note:** This deletes messages via truncation. Interruptions may corrupt your mailbox.

4.16 namedutils - Lightweight containers

The `namedutils` module defines two lightweight container types: `namedtuple` and `namedlist`. Both are subtypes of built-in sequence types, which are very fast and efficient. They simply add named attribute accessors for specific indexes within themselves.

The `namedtuple` is identical to the built-in `collections.namedtuple`, with a couple of enhancements, including a `__repr__` more suitable to inheritance.

The `namedlist` is the mutable counterpart to the `namedtuple`, and is much faster and lighter-weight than full-blown `object`. Consider this if you’re implementing nodes in a tree, graph, or other mutable data structure. If you want an even skinnier approach, you’ll probably have to look to C.

```python
boltons.namedutils.namedlist(typename, field_names, verbose=False, rename=False)
```

Returns a new subclass of list with named fields.

```python
>>> Point = namedlist('Point', ['x', 'y'])
>>> Point.__doc__
# docstring for the new class
'Point(x, y)'
>>> p = Point(11, y=22)
# instantiate with pos args or keywords
>>> p[0] + p[1]
# indexable like a plain list
33
>>> x, y = p
# unpack like a regular list
```

(continues on next page)
boltons Documentation, Release 20.1.0

boltons.namedutils.namedtuple**(typename, field_names, verbose=False, rename=False)**

Returns a new subclass of tuple with named fields.

```python
>>> Point = namedtuple('Point', ['x', 'y'])
>>> Point.__doc__
'docstring for the new class'
>>> p = Point(11, y=22)
>>> p[x] + p[y] # fields also accessible by name
33
>>> d = p._asdict() # convert to a dictionary
>>> d['x']
11
>>> Point(**d) # convert from a dictionary
Point(x=11, y=22)
>>> p._replace(x=100) # _replace() is like str.replace() but...
Point(x=100, y=22)
```

4.17 pathutils - Filesystem fun

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.

boltons.pathutils.augpath**(path, suffix='', prefix='', ext=None, base=None, dpath=None, multi-dot=False)**

Augment a path by modifying its components.

Creates 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 base-name 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 directory
- **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

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

Example

```python
>>> 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'
```

boltons.pathutils.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**

```python
>>> 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')
```

### boltons.pathutils.expandpath(*path*)

Shell-like expansion of environment variables and tilde home directory.

**Parameters** *path* (*str* / *PathLike*) – the path to expand

**Returns** expanded path

**Return type** *str*

**Example**

```python
>>> import os
>>> os.environ['SPAM'] = 'eggs'
>>> assert expandpath('~/$SPAM') == expanduser('~/eggs')
>>> assert expandpath('foo') == 'foo'
```

4.18 **queueutils - Priority queues**

Python comes with a many great data structures, from *dict* to *collections.deque*, and no shortage of serviceable algorithm implementations, from *sorted()* to *bisect*. But priority queues are curiously relegated to an example documented in *heappq*. Even there, the approach presented is not full-featured and object-oriented. There is a built-in priority queue, *Queue.PriorityQueue*, but in addition to its austere API, it carries the double-edged sword of thread-safety, making it fine for multi-threaded, multi-consumer applications, but high-overhead for cooperative/single-threaded use cases.

The *queueutils* module currently provides two Queue implementations: *HeapPriorityQueue*, based on a heap, and *SortedPriorityQueue*, based on a sorted list. Both use a unified API based on *BasePriorityQueue* to facilitate testing the slightly different performance characteristics on various application use cases.

```python
>>> pq = PriorityQueue()
>>> pq.add('low priority task', 0)
>>> pq.add('high priority task', 2)
>>> pq.add('medium priority task 1', 1)
>>> pq.add('medium priority task 2', 1)
>>> len(pq)
4
>>> pq.pop()
```
boltons Documentation, Release 20.1.0

(continued from previous page)

```python
'high priority task'
>>> pq.peek()
'medium priority task 1'
>>> len(pq)
3
```

```python
boltons.queueutils.PriorityQueue
alias of boltons.queueutils.SortedPriorityQueue

class boltons.queueutils.BasePriorityQueue(**kw)
   The abstract base class for the other PriorityQueues in this module. Override the _backend_type class attribute, as well as the _push_entry() and _pop_entry() staticmethods for custom subclass behavior. (Don't forget to use staticmethod()).

   Parameters priority_key (callable) – A function that takes priority as passed in by add() and returns a real number representing the effective priority.

add(task, priority=None)
   Add a task to the queue, or change the task’s priority if task is already in the queue. task can be any hashable object, and priority defaults to 0. Higher values representing higher priority, but this behavior can be controlled by setting priority_key in the constructor.

peek (default=_REMOVED)
   Read the next value in the queue without removing it. Returns default on an empty queue, or raises KeyError if default is not set.

pop (default=_REMOVED)
   Remove and return the next value in the queue. Returns default on an empty queue, or raises KeyError if default is not set.

remove (task)
   Remove a task from the priority queue. Raises KeyError if the task is absent.

class boltons.queueutils.HeapPriorityQueue(**kw)
   A priority queue inherited from BasePriorityQueue, backed by a list and based on the heapq.heappop() and heapq.heappush() functions in the built-in heapq module.

class boltons.queueutils.SortedPriorityQueue(**kw)
   A priority queue inherited from BasePriorityQueue, based on the bisect.insort() approach for in-order insertion into a sorted list.
```

## 4.19 setutils - IndexedSet type

The set type brings the practical expressiveness of set theory to Python. It has a very rich API overall, but lacks a couple of fundamental features. For one, sets are not ordered. On top of this, sets are not indexable, i.e., `my_set[8]` will raise an TypeError. The IndexedSet type remedies both of these issues without compromising on the excellent complexity characteristics of Python’s built-in set implementation.

```python
class boltons.setutils.IndexedSet (other=None)
   IndexedSet is a collections.MutableSet that maintains insertion order and uniqueness of inserted elements. It’s a hybrid type, mostly like an OrderedSet, but also list-like, in that it supports indexing and slicing.

   Parameters other (iterable) – An optional iterable used to initialize the set.
```
```python
>>> x = IndexedSet(list(range(4)) + list(range(8)))
>>> x
IndexedSet([0, 1, 2, 3, 4, 5, 6, 7])
>>> x - set(range(2))
IndexedSet([2, 3, 4, 5, 6, 7])
>>> x[-1]
7
>>> fcr = IndexedSet('freecreditreport.com')
>>> ''.join(fcr[:fcr.index('.')])
'freecreditpo'
```

Standard set operators and interoperation with `set` are all supported:

```python
>>> fcr & set('cash4gold.com')
IndexedSet(['c', 'd', 'o', '.', 'm'])
```

As you can see, the `IndexedSet` is almost like a `UniqueList`, retaining only one copy of a given value, in the order it was first added. For the curious, the reason why `IndexedSet` does not support setting items based on index (i.e., `__setitem__()`), consider the following dilemma:

```python
my_indexed_set = ['A', 'B', 'C', 'D']
my_indexed_set[2] = 'A'
```

At this point, a set requires only one `A`, but a `list` would overwrite `C`. Overwriting `C` would change the length of the list, meaning that `my_indexed_set[2]` would not be `A`, as expected with a `list`, but rather `D`. So, no `__setitem__()`. Otherwise, the API strives to be as complete a union of the `list` and `set` APIs as possible.

- `add(item)` → add item to the set
- `clear()` → empty the set
- `count(val)` → count number of instances of value (0 or 1)
- `difference(*others)` → get a new set with elements not in others
- `difference_update(*others)` → discard self.intersection(*others)
- `discard(item)` → discard item from the set (does not raise)
- `classmethod from_iterable(it)` → create a set from an iterable
- `index(val)` → get the index of a value, raises if not present
- `intersection(*others)` → get a set with overlap of this and others
- `intersection_update(*others)` → discard self.difference(*others)
- `issubset(other)` → return True if other contains this set
- `issuperset(other)` → return True if set contains other
- `iter_difference(*others)` → iterate over elements not in others
- `iter_intersection(*others)` → iterate over elements also in others
- `iter_slice(start, stop, step=None)` → iterate over a slice of the set
- `pop(index)` → remove the item at a given index (-1 by default)
remove(item) → remove item from the set, raises if not present

reverse() → reverse the contents of the set in-place

sort() → sort the contents of the set in-place

symmetric_difference(*others) → XOR set of this and others

symmetric_difference_update(other) → in-place XOR with other

union(*others) → return a new set containing this set and others

update(*others) → add values from one or more iterables

boltons.setutils.complement(wrapped)

Given a set, convert it to a complement set.

Whereas a set keeps track of what it contains, a complement set keeps track of what it does not contain. For example, look what happens when we intersect a normal set with a complement set:

```python
>>> list(set(range(5)) & complement(set([2, 3])))
[0, 1, 4]
```

We get the everything in the left that wasn’t in the right, because intersecting with a complement is the same as subtracting a normal set.

Parameters wrapped(set) – A set or any other iterable which should be turned into a complement set.

All set methods and operators are supported by complement sets, between other complement()-wrapped sets and/or regular set objects.

Because a complement set only tracks what elements are not in the set, functionality based on set contents is unavailable: len(), iter() (and for loops), and .pop(). But a complement set can always be turned back into a regular set by complementing it again:

```python
>>> s = set(range(5))
>>> complement(complement(s)) == s
True
```

Note: An empty complement set corresponds to the concept of a universal set from mathematics.

Many uses of sets can be expressed more simply by using a complement. Rather than trying to work out in your head the proper way to invert an expression, you can just throw a complement on the set. Consider this example of a name filter:

```python
>>> class NamesFilter(object):
...     def __init__(self, allowed):
...         self._allowed = allowed
...     def filter(self, names):
...         return [name for name in names if name in self._allowed]

>>> NamesFilter(set(['alice', 'bob'])).filter(['alice', 'bob', 'carol'])
['alice', 'bob']
```

What if we want to just express “let all the names through”?

We could try to enumerate all of the expected names:
```
NamesFilter({'alice', 'bob', 'carol'})
```

But this is very brittle – what if at some point over this object is changed to filter `['alice', 'bob', 'carol', 'dan']`?

Even worse, what about the poor programmer who next works on this piece of code? They cannot tell whether the purpose of the large allowed set was “allow everything”, or if ‘dan’ was excluded for some subtle reason.

A complement set lets the programmer intention be expressed succinctly and directly:

```
NamesFilter(complement(set()))
```

Not only is this code short and robust, it is easy to understand the intention.

## 4.20 socketutils - socket wrappers

At its heart, Python can be viewed as an extension of the C programming language. Springing from the most popular systems programming language has made Python itself a great language for systems programming. One key to success in this domain is Python’s very serviceable `socket` module and its `socket.socket` type.

The `socketutils` module provides natural next steps to the `socket` builtin: straightforward, tested building blocks for higher-level protocols.

The `BufferedSocket` wraps an ordinary socket, providing a layer of intuitive buffering for both sending and receiving. This facilitates parsing messages from streams, i.e., all sockets with type `SOCK_STREAM`. The `BufferedSocket` enables receiving until the next relevant token, up to a certain size, or until the connection is closed. For all of these, it provides consistent APIs to size limiting, as well as timeouts that are compatible with multiple concurrency paradigms. Use it to parse the next one-off text or binary socket protocol you encounter.

This module also provides the `NetstringSocket`, a pure-Python implementation of the Netstring protocol, built on top of the `BufferedSocket`, serving as a ready-made, production-grade example.

Special thanks to Kurt Rose for his original authorship and all his contributions on this module. Also thanks to Daniel J. Bernstein, the original author of Netstring.

### 4.20.1 BufferedSocket

```python
class boltons.socketutils.BufferedSocket (sock, timeout=_UNSET, maxsize=32768, recv_size=_UNSET)
```

Mainly provides `recv_until` and `recv_size`. `recv`, `send`, `sendall`, and `peek` all function as similarly as possible to the built-in socket API.

This type has been tested against both the built-in socket type as well as those from gevent and eventlet. It also features support for sockets with timeouts set to 0 (aka nonblocking), provided the caller is prepared to handle the EWOULDBLOCK exceptions.

**Parameters**

- `sock (socket)` – The connected socket to be wrapped.
- `timeout (float)` – The default timeout for sends and recvs, in seconds. Set to `None` for no timeout, and 0 for nonblocking. Defaults to `sock`'s own timeout if already set, and 10 seconds otherwise.
- `maxsize (int)` – The default maximum number of bytes to be received into the buffer before it is considered full and raises an exception. Defaults to 32 kilobytes.
• `recvsize (int) – The number of bytes to recv for every lower-level socket.recv() call. Defaults to maxsize.``

`recvsize` and `maxsize` can both be overridden on individual socket operations.

All `recv` methods return bytestrings (bytes) and can raise `socket.error`. `Timeout`, `ConnectionClosed`, and `MessageTooLong` all inherit from `socket.error` and exist to provide better error messages. Received bytes are always buffered, even if an exception is raised. Use `BufferedSocket.getrecvbuffer()` to retrieve partial recvs.

BufferedSocket does not replace the built-in socket by any means. While the overlapping parts of the API are kept parallel to the built-in `socket.socket`, BufferedSocket does not inherit from `socket`, and most socket functionality is only available on the underlying socket. `socket.getpeername()`, `socket.getsockname()`, `socket.fileno()`, and others are only available on the underlying socket that is wrapped. Use the `BufferedSocket.sock` attribute to access it. See the examples for more information on how to use BufferedSockets with built-in sockets.

The BufferedSocket is threadsafe, but consider the semantics of your protocol before accessing a single socket from multiple threads. Similarly, once the BufferedSocket is constructed, avoid using the underlying socket directly. Only use it for operations unrelated to messages, e.g., `socket.getpeername()`.

`buffer (data)`
Buffer `data` bytes for the next send operation.

`close ()`
Closes the wrapped socket, and empties the internal buffers. The send buffer is not flushed automatically, so if you have been calling `buffer()`, be sure to call `flush()` before calling this method. After calling this method, future socket operations will raise `socket.error`.

`family`
A passthrough to the wrapped socket’s family. BufferedSocket supports all widely-used families, so this read-only attribute can be one of `socket.AF_INET` for IP, `socket.AF_INET6` for IPv6, and `socket.AF_UNIX` for UDS.

`fileno ()`
Returns the file descriptor of the wrapped socket. -1 if it has been closed on this end.

Note that this makes the BufferedSocket selectable, i.e., usable for operating system event loops without any external libraries. Keep in mind that the operating system cannot know about data in BufferedSocket’s internal buffer. Exercise discipline with calling `recv*` functions.

`flush ()`
Send the contents of the internal send buffer.

`getpeername ()`
Convenience function to return the remote address to which the wrapped socket is connected. See `socket.getpeername()` for more details.

`getrecvbuffer ()`
Returns the receive buffer bytestring (rbuf).

`getsendbuffer ()`
Returns a copy of the send buffer list.

`getsockname ()`
Convenience function to return the wrapped socket’s own address. See `socket.getsockname()` for more details.

`getsockopt (level, optname, buflen=None)`
Convenience function passing through to the wrapped socket’s `socket.getsockopt()`.
peek(size, timeout=_UNSET)
Returns size bytes from the socket and/or internal buffer. Bytes are retained in BufferedSocket’s internal recv buffer. To only see bytes in the recv buffer, use getrecvbuffer().

Parameters
- size(int) – The exact number of bytes to peek at
- timeout(float) – The timeout for this operation. Can be 0 for nonblocking and None for no timeout. Defaults to the value set in the constructor of BufferedSocket.

If the appropriate number of bytes cannot be fetched from the buffer and socket before timeout expires, then a Timeout will be raised. If the connection is closed, a ConnectionClosed will be raised.

proto
A passthrough to the wrapped socket’s protocol. The proto attribute is very rarely used, so it’s always 0, meaning “the default” protocol. Pretty much all the practical information is in type and family, so you can go back to never thinking about this.

recv(size, flags=0, timeout=_UNSET)
Returns up to size bytes, using the internal buffer before performing a single socket.recv() operation.

Parameters
- size(int) – The maximum number of bytes to receive.
- flags(int) – Kept for API compatibility with sockets. Only the default, 0, is valid.
- timeout(float) – The timeout for this operation. Can be 0 for nonblocking and None for no timeout. Defaults to the value set in the constructor of BufferedSocket.

If the operation does not complete in timeout seconds, a Timeout is raised. Much like the built-in socket.socket, if this method returns an empty string, then the socket is closed and recv buffer is empty. Further calls to recv will raise socket.error.

recv_close(timeout=_UNSET, maxsize=_UNSET)
Receive until the connection is closed, up to maxsize bytes. If more than maxsize bytes are received, raises MessageTooLong.

recv_size(size, timeout=_UNSET)
Read off of the internal buffer, then off the socket, until size bytes have been read.

Parameters
- size(int) – number of bytes to read before returning.
- timeout(float) – The timeout for this operation. Can be 0 for nonblocking and None for no timeout. Defaults to the value set in the constructor of BufferedSocket.

If the appropriate number of bytes cannot be fetched from the buffer and socket before timeout expires, then a Timeout will be raised. If the connection is closed, a ConnectionClosed will be raised.

recv_until(delimiter, timeout=_UNSET, maxsize=_UNSET, with_delimiter=False)
Receive until delimiter is found, maxsize bytes have been read, or timeout is exceeded.

Parameters
- delimiter(bytes) – One or more bytes to be searched for in the socket stream.
- timeout(float) – The timeout for this operation. Can be 0 for nonblocking and None for no timeout. Defaults to the value set in the constructor of BufferedSocket.
- maxsize(int) – The maximum size for the internal buffer. Defaults to the value set in the constructor.
• **with_delimiter (bool)** – Whether or not to include the delimiter in the output. False by default, but True is useful in cases where one is simply forwarding the messages.

recv_until will raise the following exceptions:

• **Timeout** if more than timeout seconds expire.
• **ConnectionClosed** if the underlying socket is closed by the sending end.
• **MessageTooLong** if the delimiter is not found in the first maxsize bytes.
• **socket.error** if operating in nonblocking mode (timeout equal to 0), or if some unexpected socket error occurs, such as operating on a closed socket.

**send (data, flags=0, timeout=_UNSET)**

Send the contents of the internal send buffer, as well as data, to the receiving end of the connection. Returns the total number of bytes sent. If no exception is raised, all of data was sent and the internal send buffer is empty.

**Parameters**

• **data (bytes)** – The bytes to send.
• **flags (int)** – Kept for API compatibility with sockets. Only the default 0 is valid.
• **timeout (float)** – The timeout for this operation. Can be 0 for nonblocking and None for no timeout. Defaults to the value set in the constructor of BufferedSocket.

Will raise **Timeout** if the send operation fails to complete before timeout. In the event of an exception, use **BufferedSocket.getsendbuffer()** to see which data was unsent.

**sendall (data, flags=0, timeout=_UNSET)**

A passthrough to **send()**, retained for parallelism to the socket.socket API.

**setmaxsize (maxsize)**

Set the default maximum buffer size maxsize for future operations, in bytes. Does not truncate the current buffer.

**setsockopt (level, optname, value)**

Convenience function passing through to the wrapped socket’s socket.setsockopt().

**settimeout (timeout)**

Set the default timeout for future operations, in seconds.

**shutdown (how)**

Convenience method which passes through to the wrapped socket’s shutdown(). Semantics vary by platform, so no special internal handling is done with the buffers. This method exists to facilitate the most common usage, wherein a full shutdown is followed by a close(). Developers requiring more support, please open an issue.

**type**

A passthrough to the wrapped socket’s type. Valid usages should only ever see socket.SOCK_STREAM.

### Exceptions

These are a few exceptions that derive from **socket.error** and provide clearer code and better error messages.

**exception boltons.socketutils.Error**

A subclass of **socket.error** from which all other socketutils exceptions inherit.

When using **BufferedSocket** and other socketutils types, generally you want to catch one of the specific exception types below, or **socket.error**.
exception boltons.socketutils.Timeout (timeout, extra="")
    Inheriting from socket.timeout, Timeout is used to indicate when a socket operation did not complete within the time specified. Raised from any of BufferedSocket’s recv methods.

exception boltons.socketutils.ConnectionClosed
    Raised when receiving and the connection is unexpectedly closed from the sending end. Raised from BufferedSocket’s peek(), recv_until(), and recv_size(), and never from its recv() or recv_close().

exception boltons.socketutils.MessageTooLong (bytes_read=None, delimiter=None)
    Raised from BufferedSocket.recv_until() and BufferedSocket.recv_closed() when more than maxsize bytes are read without encountering the delimiter or a closed connection, respectively.

4.20.2 Netstring

class boltons.socketutils.NetstringSocket (sock, timeout=10, maxsize=32768)
    Reads and writes using the netstring protocol.


Netstring Exceptions

These are a few higher-level exceptions for Netstring connections.

exception boltons.socketutils.NetstringProtocolError
    Base class for all of socketutils’ Netstring exception types.

exception boltons.socketutils.NetstringInvalidSize (msg)
    NetstringInvalidSize is raised when the : delimited size prefix of the message does not contain a valid integer.

    Message showing valid size:

    5:hello,

    Here the 5 is the size. Anything in this prefix position that is not parsable as a Python integer (i.e., int) will raise this exception.

exception boltons.socketutils.NetstringMessageTooLong (size, maxsize)
    NetstringMessageTooLong is raised when the size prefix contains a valid integer, but that integer is larger than the NetstringSocket’s configured maxsize.

    When this exception is raised, it’s recommended to simply close the connection instead of trying to recover.

4.21 statsutils - Statistics fundamentals

statsutils provides tools aimed primarily at descriptive statistics for data analysis, such as mean() (average), median(), variance(), and many others.

The Stats type provides all the main functionality of the statsutils module. A Stats object wraps a given dataset, providing all statistical measures as property attributes. These attributes cache their results, which allows efficient computation of multiple measures, as many measures rely on other measures. For example, relative standard deviation (Stats.rel_std_dev) relies on both the mean and standard deviation. The Stats object caches those results so no rework is done.

The Stats type’s attributes have module-level counterparts for convenience when the computation reuse advantages do not apply.
>>> stats = Stats(range(42))
>>> stats.mean
20.5
>>> mean(range(42))
20.5

Statistics is a large field, and statsutils is focused on a few basic techniques that are useful in software. The following is a brief introduction to those techniques. For a more in-depth introduction, Statistics for Software, an article I wrote on the topic. It introduces key terminology vital to effective usage of statistics.

### 4.21.1 Statistical moments

Python programmers are probably familiar with the concept of the mean or average, which gives a rough quantitative middle value by which a sample can be generalized. However, the mean is just the first of four moment-based measures by which a sample or distribution can be measured.

The four Standardized moments are:

1. Mean - `mean()` - theoretical middle value
2. Variance - `variance()` - width of value dispersion
3. Skewness - `skewness()` - symmetry of distribution
4. Kurtosis - `kurtosis()` - “peakiness” or “long-tailed”-ness

For more information check out the Moment article on Wikipedia.

Keep in mind that while these moments can give a bit more insight into the shape and distribution of data, they do not guarantee a complete picture. Wildly different datasets can have the same values for all four moments, so generalize wisely.

### 4.21.2 Robust statistics

Moment-based statistics are notorious for being easily skewed by outliers. The whole field of robust statistics aims to mitigate this dilemma. statsutils also includes several robust statistical methods:

- Median - The middle value of a sorted dataset
- Trimean - Another robust measure of the data’s central tendency
- Median Absolute Deviation (MAD) - A robust measure of variability, a natural counterpart to `variance()`.
- Trimming - Reducing a dataset to only the middle majority of data is a simple way of making other estimators more robust.

### 4.21.3 Online and Offline Statistics

Unrelated to computer networking, online statistics involve calculating statistics in a streaming fashion, without all the data being available. The Stats type is meant for the more traditional offline statistics when all the data is available. For pure-Python online statistics accumulators, look at the Lithoxyl system instrumentation package.

```python
class boltons.statsutils.Stats (data, default=0.0, use_copy=True, is_sorted=False)
```

The Stats type is used to represent a group of unordered statistical datapoints for calculations such as mean, median, and variance.

**Parameters**
• **data** (*list*) – List or other iterable containing numeric values.

• **default** (*float*) – A value to be returned when a given statistical measure is not defined. 0.0 by default, but `float('nan')` is appropriate for stricter applications.

• **use_copy** (*bool*) – By default Stats objects copy the initial data into a new list to avoid issues with modifications. Pass `False` to disable this behavior.

• **is_sorted** (*bool*) – Presorted data can skip an extra sorting step for a little speed boost. Defaults to `False`.

**clear_cache()**
Stats objects automatically cache intermediary calculations that can be reused. For instance, accessing the `std_dev` attribute after the `variance` attribute will be significantly faster for medium-to-large datasets.

If you modify the object by adding additional data points, call this function to have the cached statistics recomputed.

**count**
The number of items in this Stats object. Returns the same as `len()` on a Stats object, but provided for pandas terminology parallelism.

**describe** (*quantiles=None, format=None*)
Provides standard summary statistics for the data in the Stats object, in one of several convenient formats.

**Parameters**

- **quantiles** (*list*) – A list of numeric values to use as quantiles in the resulting summary. All values must be 0.0-1.0, with 0.5 representing the median. Defaults to `[0.25, 0.5, 0.75]`, representing the standard quartiles.

- **format** (*str*) – Controls the return type of the function, with one of three valid values: "dict" gives back a `dict` with the appropriate keys and values. "list" is a list of key-value pairs in an order suitable to pass to an OrderedDict or HTML table. "text" converts the values to text suitable for printing, as seen below.

Here is the information returned by a default `describe`, as presented in the "text" format:

```python
>>> stats = Stats(range(1, 8))
>>> print(stats.describe(format='text'))
count: 7
mean: 4.0
std_dev: 2.0
mad: 2.0
min: 1
0.25: 2.5
0.5: 4
0.75: 5.5
max: 7
```

For more advanced descriptive statistics, check out my blog post on the topic Statistics for Software.

**format_histogram** (*bins=None, **kw*)
Produces a textual histogram of the data, using fixed-width bins, allowing for simple visualization, even in console environments.

```python
>>> data = list(range(20)) + list(range(5, 15)) + [10]
>>> print(Stats(data).format_histogram(width=30))
0.0: 5 ############
4.4: 8 ###############
```

(continues on next page)
In this histogram, five values are between 0.0 and 4.4, eight are between 4.4 and 8.9, and two values lie between 17.8 and the max.

You can specify the number of bins, or provide a list of bin boundaries themselves. If no bins are provided, as in the example above, Freedman’s algorithm for bin selection is used.

Parameters

- **bins** *(int)* – Maximum number of bins for the histogram. Also accepts a list of floating-point bin boundaries. If the minimum boundary is still greater than the minimum value in the data, that boundary will be implicitly added. Defaults to the bin boundaries returned by Freedman’s algorithm.

- **bin_digits** *(int)* – Number of digits to round each bin to. Note that bins are always rounded down to avoid clipping any data. Defaults to 1.

- **width** *(int)* – Integer number of columns in the longest line in the histogram. Defaults to console width on Python 3.3+, or 80 if that is not available.

- **format_bin** *(callable)* – Called on each bin to create a label for the final output. Use this function to add units, such as “ms” for milliseconds.

Should you want something more programmatically reusable, see the `get_histogram_counts()` method, the output of which is used by `format_histogram`. The `describe()` method is another useful summarization method, albeit less visual.

**get_histogram_counts** *(bins=None, **kw)*

Produces a list of (bin, count) pairs comprising a histogram of the Stats object’s data, using fixed-width bins. See `Stats.format_histogram()` for more details.

Parameters

- **bins** *(int)* – maximum number of bins, or list of floating-point bin boundaries. Defaults to the output of Freedman’s algorithm.

- **bin_digits** *(int)* – Number of digits used to round down the bin boundaries. Defaults to 1.

The output of this method can be stored and/or modified, and then passed to `statsutils.format_histogram_counts()` to achieve the same text formatting as the `format_histogram()` method. This can be useful for snapshotting over time.

**get_quantile** *(q)*

Get a quantile from the dataset. Quantiles are floating point values between 0.0 and 1.0, with 0.0 representing the minimum value in the dataset and 1.0 representing the maximum. 0.5 represents the median:

```python
>>> Stats(range(100)).get_quantile(0.5)
49.5
```

**get_zscore** *(value)*

Get the z-score for value in the group. If the standard deviation is 0, 0 inf or -inf will be returned to indicate whether the value is equal to, greater than or below the group’s mean.

**iqr**

Inter-quartile range (IQR) is the difference between the 75th percentile and 25th percentile. IQR is a
robust measure of dispersion, like standard deviation, but safer to compare between datasets, as it is less influenced by outliers.

**kurtosis**
Indicates how much data is in the tails of the distribution. The result is always positive, with the normal “bell-curve” distribution having a kurtosis of 3.


See the module docstring for more about statistical moments.

**mad**
Median Absolute Deviation is a robust measure of statistical dispersion: [http://en.wikipedia.org/wiki/Median_absolute_deviation](http://en.wikipedia.org/wiki/Median_absolute_deviation)

**max**
The maximum value present in the data.

**mean**
The arithmetic mean, or “average”. Sum of the values divided by the number of values.

**median**
The median is either the middle value or the average of the two middle values of a sample. Compared to the mean, it’s generally more resilient to the presence of outliers in the sample.

**median_abs_dev**
Median Absolute Deviation is a robust measure of statistical dispersion: [http://en.wikipedia.org/wiki/Median_absolute_deviation](http://en.wikipedia.org/wiki/Median_absolute_deviation)

**min**
The minimum value present in the data.

**pearson_type**

**rel_std_dev**
Standard deviation divided by the absolute value of the average.


**skewness**
Indicates the asymmetry of a curve. Positive values mean the bulk of the values are on the left side of the average and vice versa.


See the module docstring for more about statistical moments.

**std_dev**
Standard deviation. Square root of the variance.

**trim_relative** *(amount=0.15)*
A utility function used to cut a proportion of values off each end of a list of values. This has the effect of limiting the effect of outliers.

**Parameters**

| amount (float) | A value between 0.0 and 0.5 to trim off of each side of the data. |

**trimean**
The trimean is a robust measure of central tendency, like the median, that takes the weighted average of the median and the upper and lower quartiles.

**variance**
Variance is the average of the squares of the difference between each value and the mean.
boltons.statsutils.describe(data, quantiles=None, format=None)
A convenience function to get standard summary statistics useful for describing most data. See Stats.describe() for more details.
```python
>>> print(describe(range(7), format='text'))
    count: 7
    mean: 3.0
    std_dev: 2.0
    mad: 2.0
    min: 0
    0.25: 1.5
    0.5: 3.0
    0.75: 4.5
    max: 6
```
See Stats.format_histogram() for another very useful summarization that uses textual visualization.

boltons.statsutils.format_histogram_counts(bin_counts, width=None, format_bin=None)
The formatting logic behind Stats.format_histogram(), which takes the output of Stats.get_histogram_counts(), and passes them to this function.

Parameters
- **bin_counts** (list) – A list of bin values to counts.
- **width** (int) – Number of character columns in the text output, defaults to 80 or console width in Python 3.3+.
- **format_bin** (callable) – Used to convert bin values into string labels.

boltons.statsutils.iqr(data, default=0.0)
Inter-quartile range (IQR) is the difference between the 75th percentile and 25th percentile. IQR is a robust measure of dispersion, like standard deviation, but safer to compare between datasets, as it is less influenced by outliers.
```python
>>> iqr([1, 2, 3, 4, 5])
2
>>> iqr(range(1001))
500
```

boltons.statsutils.kurtosis(data, default=0.0)
Indicates how much data is in the tails of the distribution. The result is always positive, with the normal “bell-curve” distribution having a kurtosis of 3.
http://en.wikipedia.org/wiki/Kurtosis
See the module docstring for more about statistical moments.
```python
>>> kurtosis(range(9))
1.99125
```
With a kurtosis of 1.99125, [0, 1, 2, 3, 4, 5, 6, 7, 8] is more centrally distributed than the normal curve.

boltons.statsutils.mean(data, default=0.0)
The arithmetic mean, or “average”. Sum of the values divided by the number of values.
```python
>>> mean(range(20))
9.5
>>> mean(list(range(19)) + [949]) # 949 is an arbitrary outlier
56.0
```
boltons.statsutils.median \((data, default=0.0)\)

The median is either the middle value or the average of the two middle values of a sample. Compared to the mean, it’s generally more resilient to the presence of outliers in the sample.

```python
>>> median([2, 1, 3])
2
>>> median(range(97))
48
>>> median(list(range(96)) + [1066])  # 1066 is an arbitrary outlier
48
```

boltons.statsutils.median_abs_dev \((data, default=0.0)\)

Median Absolute Deviation is a robust measure of statistical dispersion: http://en.wikipedia.org/wiki/Median_absolute_deviation

```python
>>> median_abs_dev(range(97))
24.0
```

boltons.statsutils.pearson_type \((data, default=0.0)\)

boltons.statsutils.rel_std_dev \((data, default=0.0)\)

Standard deviation divided by the absolute value of the average.


```python
>>> print('%.3f' % rel_std_dev(range(97)))
0.583
```

boltons.statsutils.skewness \((data, default=0.0)\)

Indicates the asymmetry of a curve. Positive values mean the bulk of the values are on the left side of the average and vice versa.


See the module docstring for more about statistical moments.

```python
>>> skewness(range(97))  # symmetrical around 48.0
0.0
>>> left_skewed = skewness(list(range(97)) + list(range(10)))
>>> right_skewed = skewness(list(range(97)) + list(range(87, 97)))
>>> round(left_skewed, 3), round(right_skewed, 3)
(0.114, -0.114)
```

boltons.statsutils.std_dev \((data, default=0.0)\)

Standard deviation. Square root of the variance.

```python
>>> std_dev(range(97))
28.0
```

boltons.statsutils.trimean \((data, default=0.0)\)

The trimean is a robust measure of central tendency, like the median, that takes the weighted average of the median and the upper and lower quartiles.

```python
>>> trimean([2, 1, 3])
2.0
>>> trimean(range(97))
48.0
>>> trimean(list(range(96)) + [1066])  # 1066 is an arbitrary outlier
48.0
```
boltons.statsutils.variance (data, default=0.0)
Variance is the average of the squares of the difference between each value and the mean.

```python
>>> variance(range(97))
784.0
```

## 4.22 strutils - Text manipulation

So much practical programming involves string manipulation, which Python readily accommodates. Still, there are dozens of basic and common capabilities missing from the standard library, several of them provided by strutils.

boltons.strutils.camel2under (camel_string)
Converts a cameldcase string to underscored. Useful for turning a class name into a function name.

```python
>>> camel2under('BasicParseTest')
'basic_parse_test'
```

boltons.strutils.under2camel (under_string)
Converts an underscored string to cameldcase. Useful for turning a function name into a class name.

```python
>>> under2camel('complex_tokenizer')
'ComplexTokenizer'
```

boltons.strutils.slugify (text, delim='_', lower=True, ascii=False)
A basic function that turns text full of scary characters (i.e., punctuation and whitespace), into a relatively safe lowercased string separated only by the delimiter specified by `delim`, which defaults to `_`.

The `ascii` convenience flag will `asciify()` the slug if you require ascii-only slugs.

```python
>>> slugify('First post! Hi!!!!~1 ') 'first_post_hi_1'

>>> slugify("Kurt Gödel's pretty cool.", ascii=True) == b'kurt_goedel_s__pretty_cool'
True
```

boltons.strutils.split_punct_ws (text)
While `str.split()` will split on whitespace, `split_punct_ws()` will split on punctuation and whitespace. This used internally by `slugify()`, above.

```python
>>> split_punct_ws('First post! Hi!!!!~1 ') ['First', 'post', 'Hi', '1']
```

boltons.strutils.unit_len (sized_iterable, unit_noun='item')
Returns a plain-English description of an iterable’s `len()`, conditionally pluralized with `cardinalize()`, detailed below.

```python
>>> print(unit_len(range(10), 'number'))
10 numbers
>>> print(unit_len('aeiou', 'vowel'))
5 vowels
>>> print(unit_len([], 'worry'))
No worries
```
boltons Documentation, Release 20.1.0

boltons.strutils.ordinalize(number, ext_only=False)

Turns number into its cardinal form, i.e., 1st, 2nd, 3rd, 4th, etc. If the last character isn’t a digit, it returns the string value unchanged.

Parameters

• number (int or str) – Number to be cardinalized.

• ext_only (bool) – Whether to return only the suffix. Default False.

```python
>>> print(ordinalize(1))
1st
>>> print(ordinalize(3694839230))
3694839230th
>>> print(ordinalize('hi'))
hi
>>> print(ordinalize(1515))
1515th
```

boltons.strutils.cardinalize(unit_noun, count)

Conditionally pluralizes a singular word unit_noun if count is not one, preserving case when possible.

```python
>>> vowels = 'aeiou'
>>> print(len(vowels), cardinalize('vowel', len(vowels)))
5 vowels
>>> print(3, cardinalize('Wish', 3))
3 Wishes
```

boltons.strutils.pluralize(word)

Semi-intelligently converts an English word from singular form to plural, preserving case pattern.

```python
>>> pluralize('friend')
'friends'
>>> pluralize('enemy')
'enemies'
>>> pluralize('Sheep')
'Sheep'
```

boltons.strutils.singularize(word)

Semi-intelligently converts an English plural word to its singular form, preserving case pattern.

```python
>>> singularize('chances')
'chance'
>>> singularize('Activities')
'Activity'
>>> singularize('Glasses')
'Glass'
>>> singularize('FEET')
'FOOT'
```

boltons.strutils.asciify(text, ignore=False)

Converts a unicode or bytestring, text, into a bytestring with just ascii characters. Performs basic deaccenting for all you Europhiles out there.

Also, a gentle reminder that this is a utility, primarily meant for slugification. Whenever possible, make your application work with unicode, not against it.

Parameters

• text (str or unicode) – The string to be asciified.
• **ignore** (*bool*) — Configures final encoding to ignore remaining unasciified unicode instead of replacing it.

```python
>>> asciify('Beyoncé') == b'Beyonce'
True
```

boltons.strutils.is_ascii(*text*)

Check if a unicode or bytestring, *text*, is composed of ascii characters only. Raises ValueError if argument is not text.

**Parameters**

- **text** (*str or unicode*) – The string to be checked.

```python
>>> is_ascii('Beyoncé')
False
>>> is_ascii('Beyonce')
True
```

boltons.strutils.is_uuid(*obj*, version=4)

Check the argument is either a valid UUID object or string.

**Parameters**

- **obj** (*object*) – The test target. Strings and UUID objects supported.
- **version** (*int*) – The target UUID version, set to 0 to skip version check.

```python
>>> is_uuid('e682cccc-5a4c-4ef2-9711-73f9ad1e15ea')
True
>>> is_uuid('0221f0d9-d4b9-11e5-a478-10ddb1c2feb9')
False
>>> is_uuid('0221f0d9-d4b9-11e5-a478-10ddb1c2feb9', version=1)
True
```

boltons.strutils.html2text(*html*)

Strips tags from HTML text, returning markup-free text. Also, does a best effort replacement of entities like “&nbsp;”

```python
>>> r = html2text(u'<a href="#">Test &amp; <em>(\u0394\u03b7\u03bc\u03ce)</em> →</a>')
>>> r == u'Test &(<u0394\u03b7\u03bc\u03ce)
True
```

boltons.strutils.strip_ansi(*text*)

Strips ANSI escape codes from *text*. Useful for the occasional time when a log or redirected output accidentally captures console color codes and the like.

```python
>>> strip_ansi([0m[1;36mart[46;34m0])
'art'
```

The test above is an excerpt from ANSI art on sixteencolors.net. This function does not interpret or render ANSI art, but you can do so with ansi2img or escapes.js.

boltons.strutils.bytes2human(*nbytes*, ndigits=0)

Turns an integer value of *nbytes* into a human readable format. Set *ndigits* to control how many digits after the decimal point should be shown (default 0).

```python
>>> bytes2human(128991)
'126K'
>>> bytes2human(100001221)
```

(continues on next page)
boltons Documentation, Release 20.1.0

'95M'

```python
>>> bytes2human(0, 2)
'0.00B'
```

**boltons.strutils.find_hashtags** *(string)*

Finds and returns all hashtags in a string, with the hashmark removed. Supports full-width hashmarks for Asian languages and does not false-positive on URL anchors.

```python
>>> find_hashtags('#atag http://asite/#ananchor')
['atag']
```

find_hashtags also works with unicode hashtags.

**boltons.strutils.a10n** *(string)*

That thing where “internationalization” becomes “i18n”, what’s it called? Abbreviation? Oh wait, no: a10n. (It’s actually a form of numeronym.)

```python
>>> a10n('abbreviation')
'a10n'

>>> a10n('internationalization')
'ii18n'

>>> a10n('')
''
```

**boltons.strutils.gzip_bytes** *(bytestring, level=6)*

Turn some bytes into some compressed bytes.

```python
>>> len(gzip_bytes(b'a' * 10000))
46
```

**Parameters**

- **bytestring** *(bytes)* – Bytes to be compressed
- **level** *(int)* – An integer, 1-9, controlling the speed/compression. 1 is fastest, least compressed, 9 is slowest, but most compressed.

Note that all levels of gzip are pretty fast these days, though it’s not really a competitor in compression, at any level.

**boltons.strutils.gunzip_bytes** *(bytestring)*

The gzip module is great if you have a file or file-like object, but what if you just have bytes. StringIO is one possibility, but it’s often faster, easier, and simpler to just use this one-liner. Use this tried-and-true utility function to decompress gzip from bytes.

```python
>>> gunzip_bytes(_EMPTY_GZIP_BYTES) == b''
True

>>> gunzip_bytes(_NON_EMPTY_GZIP_BYTES).rstrip() == b'bytesahoy!'
True
```

**boltons.strutils.iter_splitlines** *(text)*

Like `str.splitlines()`, but returns an iterator of lines instead of a list. Also similar to `file.next()`, as that also lazily reads and yields lines from a file.

This function works with a variety of line endings, but as always, be careful when mixing line endings within a file.
boltons Documentation, Release 20.1.0

>>> list(iter_splitlines('\nhi
bye\n'))
['', 'hi', 'bye', '']
>>> list(iter_splitlines('\rhi\rbye\r'))
['', 'hi', 'bye', '']
>>> list(iter_splitlines(''))
[]

boltons.strutils.indent (text, margin, newline='\n', key=<type 'bool'>)
The missing counterpart to the built-in textwrap.dedent ()

Parameters

• text (str) – The text to indent.
• margin (str) – The string to prepend to each line.
• newline (str) – The newline used to rejoin the lines (default: \n)
• key (callable) – Called on each line to determine whether to indent it. Default: bool, to ensure that empty lines do not get whitespace added.

boltons.strutils.escape_shell_args (args, sep=' ', style=None)
Returns an escaped version of each string in args, according to style.

Parameters

• args (list) – A list of arguments to escape and join together
• sep (str) – The separator used to join the escaped arguments.
• style (str) – The style of escaping to use. Can be one of cmd or sh, geared toward Windows and Linux/BSD/etc., respectively. If style is None, then it is picked according to the system platform.

See args2cmd () and args2sh () for details and example output for each style.

boltons.strutils.args2cmd (args, sep=' ')
Return a shell-escaped string version of args, separated by sep, using the same rules as the Microsoft C runtime.

>>> print(args2cmd(['aa', '[bb]', "cc'cc", 'dd"dd']))
aa [bb] cc'cc dd"dd

As you can see, escaping is through backslashing and not quoting, and double quotes are the only special character. See the comment in the code for more details. Based on internal code from the subprocess module.

boltons.strutils.args2sh (args, sep=' ')
Return a shell-escaped string version of args, separated by sep, based on the rules of sh, bash, and other shells in the Linux/BSD/MacOS ecosystem.

>>> print(args2sh(['aa', '[bb]', "cc'cc", 'dd"dd']))
aa ' [bb]' "cc'""cc" 'dd"dd'

As you can see, arguments with no special characters are not escaped, arguments with special characters are quoted with single quotes, and single quotes themselves are quoted with double quotes. Double quotes are handled like any other special character.

Based on code from the pipes/shlex modules. Also note that shlex and argpase have functions to split and parse strings escaped in this manner.

boltons.strutils.parse_int_list (range_string, delim=':', range_delim=':')
Returns a sorted list of positive integers based on range_string. Reverse of format_int_list ().
Parameters

- **range_string (str)** – String of comma separated positive integers or ranges (e.g. ‘1,2,4-6,8’). Typical of a custom page range string used in printer dialogs.
- **delim (char)** – Defaults to ‘,’. Separates integers and contiguous ranges of integers.
- **range_delim (char)** – Defaults to ‘-’. Indicates a contiguous range of integers.

```python
>>> parse_int_list('1,3,5-8,10-11,15')
[1, 3, 5, 6, 7, 8, 10, 11, 15]
```

boltons.strutils.format_int_list(int_list, delim=' ', range_delim='-', delim_space=False)

Returns a sorted range string from a list of positive integers (int_list). Contiguous ranges of integers are collapsed to min and max values. Reverse of `parse_int_list()`.

Parameters

- **int_list (list)** – List of positive integers to be converted into a range string (e.g. [1,2,4,5,6,8]).
- **delim (char)** – Defaults to ‘,’. Separates integers and contiguous ranges of integers.
- **range_delim (char)** – Defaults to ‘-’. Indicates a contiguous range of integers.
- **delim_space (bool)** – Defaults to False. If True, adds a space after all delim characters.

```python
>>> format_int_list([1,3,5,6,7,8,10,11,15])
'1,3,5-8,10-11,15'
```

boltons.strutils.unwrap_text (text, ending='

')

Unwrap text, the natural complement to `textwrap.wrap()`.

```python
>>> text = "Short
  
lines
  
wrapped
small.

Another
paragraph."

>>> unwrap_text(text)
'Short lines wrapped small.
Another paragraph.'
```

Parameters

- **text** – A string to unwrap.
- **ending (str)** – The string to join all unwrapped paragraphs by. Pass None to get the list. Defaults to ‘\n’ for compatibility with Markdown and RST.

### 4.23 tableutils - 2D data structure

If there is one recurring theme in *boltons*, it is that Python has excellent datastructures that constitute a good foundation for most quick manipulations, as well as building applications. However, Python usage has grown much faster than builtin data structure power. Python has a growing need for more advanced general-purpose data structures which behave intuitively.

The *Table* class is one example. When handed one- or two-dimensional data, it can provide useful, if basic, text and HTML renditions of small to medium sized data. It also heuristically handles recursive data of various formats (lists, dicts, namedtuples, objects).

For more advanced *Table*-style manipulation check out the pandas DataFrame.
**class** `boltons.tableutils.Table` *(data=None, headers=_MISSING, metadata=None)*

This Table class is meant to be simple, low-overhead, and extensible. Its most common use would be for translation between in-memory data structures and serialization formats, such as HTML and console-ready text.

As such, it stores data in list-of-lists format, and *does not* copy lists passed in. It also reserves the right to modify those lists in a “filling” process, whereby short lists are extended to the width of the table (usually determined by number of headers). This greatly reduces overhead and processing/validation that would have to occur otherwise.

General description of headers behavior:

Headers describe the columns, but are not part of the data, however, if the `headers` argument is omitted, Table tries to infer header names from the data. It is possible to have a table with no headers, just pass in `headers=None`.

Supported inputs:
- list of list objects
- dict (list/single)
- object (list/single)
- collections.namedtuple (list/single)
- TODO: DB API cursor?
- TODO: json

Supported outputs:
- HTML
- Pretty text (also usable as GF Markdown)
- TODO: CSV
- TODO: json
- TODO: json lines

To minimize resident size, the Table data is stored as a list of lists.

**extend**(data)

Append the given data to the end of the Table.

**classmethod from_data**(data, headers=_MISSING, max_depth=1, **kwargs)

Create a Table from any supported data, heuristically selecting how to represent the data in Table format.

**Parameters**

- **data** *(object)* – Any object or iterable with data to be imported to the Table.
- **headers** *(iterable)* – An iterable of headers to be matched to the data. If not explicitly passed, headers will be guessed for certain datatypes.
- **max_depth** *(int)* – The level to which nested Tables should be created (default: 1).
- **data_type** *(InputType subclass)* – For advanced use cases, do not guess the type of the input data, use this data type instead.

**classmethod from_dict**(data, headers=_MISSING, max_depth=1, metadata=None)

Create a Table from a **dict**. Operates the same as `from_data()`, but forces interpretation of the data as a **dict**.
classmethod from_list (data, headers=_MISSING, max_depth=1, metadata=None)
Create a Table from a list. Operates the same as from_data(), but forces the interpretation of the data as a Sequence.

classmethod from_object (data, headers=_MISSING, max_depth=1, metadata=None)
Create a Table from an object. Operates the same as from_data(), but forces the interpretation of the data as an object. May be useful for some dict and list subtypes.

get_cell_html (value)
Called on each value in an HTML table. By default it simply escapes the HTML. Override this method to add additional conditions and behaviors, but take care to ensure the final output is HTML escaped.

to_html (orientation=None, wrapped=True, with_headers=True, with_newlines=True,
with_metadata=False, max_depth=1)
Render this Table to HTML. Configure the structure of Table HTML by subclassing and overriding _html_* class attributes.

Parameters

• orientation (str) – one of ‘auto’, ‘horizontal’, or ‘vertical’ (or the first letter of any of those). Default ‘auto’.

• wrapped (bool) – whether or not to include the wrapping ‘<table></table>’ tags. Default True, set to False if appending multiple Table outputs or an otherwise customized HTML wrapping tag is needed.

• with_newlines (bool) – Set to True if output should include added newlines to make the HTML more readable. Default False.

• with_metadata (bool/str) – Set to True if output should be preceded with a Table of preset metadata, if it exists. Set to special value ‘bottom’ if the metadata Table HTML should come after the main HTML output.

• max_depth (int) – Indicate how deeply to nest HTML tables before simply reverting to repr()-ing the nested data.

Returns A text string of the HTML of the rendered table.

to_text (with_headers=True, maxlen=None)
Get the Table’s textual representation. Only works well for Tables with non-recursive data.

Parameters

• with_headers (bool) – Whether to include a header row at the top.

• maxlen (int) – Max length of data in each cell.

4.24 tbutils - Tracebacks and call stacks

One of the oft-cited tenets of Python is that it is better to ask forgiveness than permission. That is, there are many cases where it is more inclusive and correct to handle exceptions than spend extra lines and execution time checking for conditions. This philosophy makes good exception handling features all the more important. Unfortunately Python’s traceback module is woefully behind the times.

The tbutils module provides two disparate but complementary feature sets:

1. With ExceptionInfo and TracebackInfo, the ability to extract, construct, manipulate, format, and serialize exceptions, tracebacks, and callstacks.

2. With ParsedException, the ability to find and parse tracebacks from captured output such as logs and stdout.
There is also the `ContextualTracebackInfo` variant of `TracebackInfo`, which includes much more information from each frame of the callstack, including values of locals and neighboring lines of code.

```python
class boltons.tbutils.ExceptionInfo(exc_type, exc_msg, tb_info)

An ExceptionInfo object ties together three main fields suitable for representing an instance of an exception: The exception type name, a string representation of the exception itself (the exception message), and information about the traceback (stored as a `TracebackInfo` object).

These fields line up with `sys.exc_info()`, but unlike the values returned by that function, ExceptionInfo does not hold any references to the real exception or traceback. This property makes it suitable for serialization or long-term retention, without worrying about formatting pitfalls, circular references, or leaking memory.

Parameters

- **exc_type** (`str`) – The exception type name.
- **exc_msg** (`str`) – String representation of the exception value.
- **tb_info** (`TracebackInfo`) – Information about the stack trace of the exception.

Like the `TracebackInfo`, ExceptionInfo is most commonly instantiated from one of its classmethods: `from_exc_info()` or `from_current()`.

```classmethod from_current()

Create an ExceptionInfo object from the current exception being handled, by way of `sys.exc_info()`. Will raise an exception if no exception is currently being handled.

```classmethod from_exc_info(exc_type, exc_value, traceback)

Create an ExceptionInfo object from the exception’s type, value, and traceback, as returned by `sys.exc_info()`. See also `from_current()`.

```get_formatted()

Returns a string formatted in the traditional Python built-in style observable when an exception is not caught. In other words, mimics `traceback.format_exception()`.

```tb_info_type

Override this in inherited types to control the `TracebackInfo` type used

alias of `TracebackInfo`

```to_dict()

Get a `dict` representation of the ExceptionInfo, suitable for JSON serialization.

```class boltons.tbutils.TracebackInfo(frames)

The TracebackInfo class provides a basic representation of a stack trace, be it from an exception being handled or just part of normal execution. It is basically a wrapper around a list of `Callpoint` objects representing frames.

Parameters frames (`list`) – A list of frame objects in the stack.

Note: TracebackInfo can represent both exception tracbacks and non-exception tracbacks (aka stack traces). As a result, there is no `TracebackInfo.from_current()`, as that would be ambiguous. Instead, call `TracebackInfo.from_frame()` without the frame argument for a stack trace, or `TracebackInfo.from_traceback()` without the tb argument for an exception traceback.

```callpoint_type

alias of `Callpoint`

```classmethod from_dict(d)

Complements `TracebackInfo.to_dict()`.
classmethod from_frame(frame=None, level=1, limit=None)
Create a new TracebackInfo frame by recurring up in the stack a max of limit times. If frame is unset, get
the frame from sys._getframe() using level.

Parameters

- **frame** (types.FrameType) – frame object from sys._getframe() or elsewhere. Defaults to result of sys.get_frame().
- **level** (int) – If frame is unset, the desired frame is this many levels up the stack from
the invocation of this method. Default 1 (i.e., caller of this method).
- **limit** (int) – max number of parent frames to extract (defaults to sys.tracebacklimit)

classmethod from_traceback(tb=None, limit=None)
Create a new TracebackInfo from the traceback tb by recurring up in the stack a max of limit times. If tb
is unset, get the traceback from the currently handled exception. If no exception is being handled, raise a
ValueError.

Parameters

- **frame** (types.TracebackType) – traceback object from sys.exc_info() or elsewhere. If absent or set to None, defaults to sys.exc_info()[2], and raises a
ValueError if no exception is currently being handled.
- **limit** (int) – max number of parent frames to extract (defaults to sys.tracebacklimit)

get_formatted()
Returns a string as formatted in the traditional Python built-in style observable when an exception is not
caught. In other words, mimics traceback.format_tb() and traceback.format_stack().

to_dict()
Returns a dict with a list of Callpoint frames converted to dicts.

class boltons.tbutils.Callpoint(module_name, module_path, func_name, lineno, lasti, line=None)
The Callpoint is a lightweight object used to represent a single entry in the code of a call stack. It stores the
code-related metadata of a given frame. Available attributes are the same as the parameters below.

Parameters

- **func_name** (str) – the function name
- **lineno** (int) – the line number
- **module_name** (str) – the module name
- **module_path** (str) – the filesystem path of the module
- **lasti** (int) – the index of bytecode execution
- **line** (str) – the single-line code content (if available)

classmethod from_current(level=1)
Creates a Callpoint from the location of the calling function.

classmethod from_frame(frame)
Create a Callpoint object from data extracted from the given frame.

classmethod from_tb(tb)
Create a Callpoint from the traceback of the current exception. Main difference with from_frame() is that
lineno and lasti come from the traceback, which is to say the line that failed in the try block, not
the line currently being executed (in the except block).
`tb_frame_str()`
Render the Callpoint as it would appear in a standard printed Python traceback. Returns a string with filename, line number, function name, and the actual code line of the error on up to two lines.

`to_dict()`
Get a dict copy of the Callpoint. Useful for serialization.

```python
class boltons.tbutils.ContextualExceptionInfo(exc_type, exc_msg, tb_info)
The ContextualTracebackInfo type is a `TracebackInfo` subtype that uses the `ContextualCallpoint` as its frame-representing primitive.
It carries with it most of the exception information required to recreate the widely recognizable “500” page for debugging Django applications.

class ContextualTracebackInfo
    alias of ContextualTracebackInfo

class ContextualCallpoint
    alias of ContextualCallpoint
```

The ContextualCallpoint is a `Callpoint` subtype with the exact same API and storing two additional values:

1. repr() outputs for local variables from the Callpoint’s scope
2. A number of lines before and after the Callpoint’s line of code

The ContextualCallpoint is used by the `ContextualTracebackInfo`.

```python
classmethod from_frame(frame)
    Identical to Callpoint.from_frame()
classmethod from_tb(tb)
    Identical to Callpoint.from_tb()
to_dict()
    Same principle as Callpoint.to_dict(), but with the added contextual values. With ContextualCallpoint.to_dict(), each frame will now be represented like:

```python
dictionary = {
    'func_name': 'print_example',
    'lineno': 0,
    'module_name': 'example_module',
    'module_path': '/home/example/example_module.pyc',
    'lasti': 0,
    'line': 'print "example"',
    'locals': {'variable': "value"},
    'pre_lines': [{'variable = "value"'},
    'post_lines': []}
```

The locals dictionary and line lists are copies and can be mutated freely.

```python
class boltons.tbutils.print_exception(etype, value, tb, limit=None, file=None)
    Print exception up to ‘limit’ stack trace entries from ‘tb’ to ‘file’.

    This differs from print_tb() in the following ways: (1) if traceback is not None, it prints a header “Traceback (most recent call last):”; (2) it prints the exception type and value after the stack trace; (3) if type is SyntaxError
```

4.24. `tbutils` - Tracebacks and call stacks
and value has the appropriate format, it prints the line where the syntax error occurred with a caret on the next line indicating the approximate position of the error.

**class boltons.tbutils.ParsedException** (*exc_type_name, exc_msg, frames=None*)

Stores a parsed traceback and exception as would be typically output by `sys.excepthook()` or `traceback.print_exception()`.

**classmethod from_string** (*tb_str*)

Parse a traceback and exception from the text `tb_str`. This text is expected to have been decoded, otherwise it will be interpreted as UTF-8.

This method does not search a larger body of text for tracebacks. If the first line of the text passed does not match one of the known patterns, a `ValueError` will be raised. This method will ignore trailing text after the end of the first traceback.

**Parameters**

- **tb_str** (*str*) – The traceback text (unicode or UTF-8 bytes)

**source_file**

The file path of module containing the function that raised the exception, or None if not available.

**to_dict()**

Get a copy as a JSON-serializable dict.

**to_string()**

Formats the exception and its traceback into the standard format, as returned by the traceback module.

```python
ParsedException.from_string(text).to_string() should yield text.
```

## 4.25 timeutils - datetime additions

Python’s `datetime` module provides some of the most complex and powerful primitives in the Python standard library. Time is nontrivial, but thankfully its support is first-class in Python. `dateutils` provides some additional tools for working with time.

Additionally, timeutils provides a few basic utilities for working with timezones in Python. The Python `datetime` module’s documentation describes how to create a `datetime`-compatible `tzinfo` subtype. It even provides a few examples.

The following module defines usable forms of the timezones in those docs, as well as a couple other useful ones, `UTC` (aka GMT) and `LocalTZ` (representing the local timezone as configured in the operating system). For timezones beyond these, as well as a higher degree of accuracy in corner cases, check out `pytz` and `dateutil`.

**boltons.timeutils.daterange** (*start, stop, step=1, inclusive=False*)

In the spirit of `range()` and `xrange()`, the `daterange` generator that yields a sequence of `date` objects, starting at `start`, incrementing by `step`, until `stop` is reached.

When `inclusive` is True, the final date may be `stop`, if `step` falls evenly on it. By default, `step` is one day. See details below for many more details.

**Parameters**

- **start** (*datetime.date*) – The starting date. The first value in the sequence.
- **stop** (*datetime.date*) – The stopping date. By default not included in return. Can be `None` to yield an infinite sequence.
- **step** (*int*) – The value to increment `start` by to reach `stop`. Can be an `int` number of days, a `datetime.timedelta`, or a tuple of integers, `(year, month, day)`. Positive and negative `step` values are supported.
• $\textbf{inclusive (bool)}$ – Whether or not the $\text{stop}$ date can be returned. $\text{stop}$ is only returned when a $\text{step}$ falls evenly on it.

```python
>>> christmas = date(year=2015, month=12, day=25)
>>> boxing_day = date(year=2015, month=12, day=26)
>>> new_year = date(year=2016, month=1, day=1)
>>> for day in daterange(christmas, new_year):
...     print(repr(day))
...     datetime.date(2015, 12, 25)
...     datetime.date(2015, 12, 26)
...     datetime.date(2015, 12, 27)
...     datetime.date(2015, 12, 28)
...     datetime.date(2015, 12, 29)
...     datetime.date(2015, 12, 30)
...     datetime.date(2015, 12, 31)
>>> for day in daterange(christmas, boxing_day):
...     print(repr(day))
...     datetime.date(2015, 12, 25)
>>> for day in daterange(christmas, boxing_day):
...     print(repr(day))
...     datetime.date(2015, 12, 25)
...     datetime.date(2015, 12, 26)
...     datetime.date(2015, 12, 27)
...     datetime.date(2015, 12, 28)
...     datetime.date(2015, 12, 29)
...     datetime.date(2015, 12, 30)
...     datetime.date(2015, 12, 31)
```

Be careful when using $\text{stop}=$None, as this will yield an infinite sequence of dates.

boltons.timeutils.$\textbf{isoparse (iso_str)}$

Parses the limited subset of ISO8601-formatted time strings as returned by $\text{datetime.datetime.isoformat()}$.

```python
>>> epoch_dt = datetime.utcfromtimestamp(0)
>>> iso_str = epoch_dt.isoformat()
>>> print(iso_str)
1970-01-01T00:00:00
>>> isoparse(iso_str)
datetime.datetime(1970, 1, 1, 0, 0)
```

For further datetime parsing, see the $\text{iso8601}$ package for strict ISO parsing and ‘$\text{dateutil}$’ package for loose parsing and more.

boltons.timeutils.$\textbf{parse_timedelta (text)}$

Robustly parses a short text description of a time period into a $\text{datetime.timedelta}$. Supports weeks, days, hours, minutes, and seconds, with or without decimal points:

- **Parameters** $\textbf{text (str)}$ – Text to parse.
- **Returns** $\text{datetime.timedelta}$
- **Raises** $\text{ValueError}$ – on parse failure.

```python
>>> parse_td('1d 2h 3.5m 0s') == timedelta(days=1, seconds=7410)
True
```

Also supports full words and whitespace.
>>> parse_td('2 weeks 1 day') == timedelta(days=15)
True

Negative times are supported, too:

>>> parse_td('-1.5 weeks 3m 20s') == timedelta(days=-11, seconds=43400)
True

boltons.timeutils.strpdate(string, format)
Parse the date string according to the format in format. Returns a date object. Internally, datetime.strptime() is used to parse the string and thus conversion specifiers for time fields (e.g. %H) may be provided; these will be parsed but ignored.

Parameters

- **string**(str) – The date string to be parsed.
- **format**(str) – The strftime-style date format string.

Returns
datetime.date

>>> strpdate('2016-02-14', '%Y-%m-%d')
datetime.date(2016, 2, 14)
>>> strpdate('26/12 (2015)', '%d/%m (%Y)')
datetime.date(2015, 12, 26)
>>> strpdate('20151231 23:59:59', '%Y%m%d %H:%M:%S')
datetime.date(2015, 12, 31)
>>> strpdate('20160101 00:00:00.001', '%Y%m%d %H:%M:%S.%f')
datetime.date(2016, 1, 1)

boltons.timeutils.total_seconds(td)
For those with older versions of Python, a pure-Python implementation of Python 2.7’s total_seconds().

Parameters
td (datetime.timedelta) – The timedelta to convert to seconds.

Returns
total number of seconds

Return type
float

>>> td = timedelta(days=4, seconds=33)
>>> total_seconds(td)
345633.0

boltons.timeutils.dt_to_timestamp(dt)
Converts from a datetime object to an integer timestamp, suitable interoperation with time.time() and other Epoch-based timestamps.

>>> abs(round(time.time() - dt_to_timestamp(datetime.utcnow()), 2))
0.0

dt_to_timestamp supports both timezone-aware and naïve datetime objects. Note that it assumes naïve datetime objects are implied UTC, such as those generated with datetime.datetime.utcnow(). If your datetime objects are local time, such as those generated with datetime.datetime.now(), first convert it using the datetime.datetime.replace() method with tzinfo= LocalTZ object in this module, then pass the result of that to dt_to_timestamp.

boltons.timeutils.relative_time(d, other=None, ndigits=0)
Get a string representation of the difference between two datetime objects or one datetime and the current time. Handles past and future times.
Parameters

- **d** (*datetime*) – The first datetime object.
- **other** (*datetime*) – An optional second datetime object. If unset, defaults to the current time as determined `datetime.utcnow()`.
- **ndigits** (*int*) – The number of decimal digits to round to, defaults to 0.

Returns

A short English-language string.

```python
>>> now = datetime.utcnow()
>>> relative_time(now, ndigits=1)
'0 seconds ago'
>>> relative_time(now - timedelta(days=1, seconds=36000), ndigits=1)
'1.4 days ago'
>>> relative_time(now + timedelta(days=7), now, ndigits=1)
'1 week from now'
```

boltons.timeutils.decimal_relative_time (*d, other=None, ndigits=0, cardinalize=True*)

Get a tuple representing the relative time difference between two `datetime` objects or one `datetime` and now.

Parameters

- **d** (*datetime*) – The first datetime object.
- **other** (*datetime*) – An optional second datetime object. If unset, defaults to the current time as determined `datetime.utcnow()`.
- **ndigits** (*int*) – The number of decimal digits to round to, defaults to 0.
- **cardinalize** (*bool*) – Whether to pluralize the time unit if appropriate, defaults to True.

Returns

A tuple of the float difference and respective unit of time, pluralized if appropriate and cardinalize is set to True.

Return type: *(float, str)*

Unlike `relative_time()`, this method’s return is amenable to localization into other languages and custom phrasing and formatting.

```python
>>> now = datetime.utcnow()
>>> decimal_relative_time(now - timedelta(days=1, seconds=36000), now)
(1.0, 'day')
>>> decimal_relative_time(now - timedelta(seconds=0.002), now, ndigits=5)
(0.002, 'seconds')
>>> decimal_relative_time(now, now - timedelta(days=900), ndigits=1)
(-2.5, 'years')
```

### 4.25.1 General time zones

By default, `datetime.datetime` objects are “naïve”, meaning they lack attached timezone information. These objects can be useful for many operations, but many operations require timezone-aware datetimes.

The two most important timezones in programming are Coordinated Universal Time (UTC) and the local timezone of the host running your code. Boltons provides two `datetime.tzinfo` subtypes for working with them:

```python
timeutils.UTC = ConstantTZInfo(name='UTC', offset=datetime.timedelta(0))
```
The `LocalTZ` type takes data available in the time module about the local timezone and makes a practical `datetime.tzinfo` to represent the timezone settings of the operating system.

For a more in-depth integration with the operating system, check out `tzlocal`. It builds on `pytz` and implements heuristics for many versions of major operating systems to provide the official `pytz tzinfo`, instead of the `LocalTZ` generalization.

```python
class boltons.timeutils.ConstantTZInfo(name='ConstantTZ', offset=datetime.timedelta(0))

A `tzinfo` subtype whose `offset` remains constant (no daylight savings).

Parameters
- `name (str)` – Name of the timezone.
- `offset (datetime.timedelta)` – Offset of the timezone.
```

### 4.25.2 US timezones

These four US timezones were implemented in the `datetime` documentation and have been reproduced here in `boltons` for convenience. More in-depth support is provided by `dateutil` and `pytz`.

```python
timeutils.Eastern = Eastern
timeutils.Central = Central
timeutils.Mountain = Mountain
timeutils.Pacific = Pacific
```

```python
class boltons.timeutils.USTimeZone(hours, reprname, stdname, dstname)

Copied directly from the Python docs, the USTimeZone is a `datetime.tzinfo` subtype used to create the `Eastern`, `Central`, `Mountain`, and `Pacific` `tzinfo` types.
```

### 4.26 typeutils - Type handling

Python’s built-in `functools` module builds several useful utilities on top of Python’s first-class function support. `typeutils` attempts to do the same for metaprogramming with types and instances.

```python
class boltons.typeutils.classproperty(fn)

Much like a `property`, but the wrapped get function is a class method. For simplicity, only read-only properties are implemented.

```python
boltons.typeutils.get_all_subclasses(cls)

Recursively finds and returns a list of all types inherited from `cls`.

```python
>>> class A(object):
...     pass
...
>>> class B(A):
...     pass
...
>>> class C(B):
...     pass
...
>>> class D(C):
...     pass
...
```

(continues on next page)
boltons Documentation, Release 20.1.0

boltons.typeutils.issubclass(subclass, baseclass)

Just like the built-in issubclass(), this function checks whether subclass is inherited from baseclass. Unlike the built-in function, this issubclass will simply return False if either argument is not suitable (e.g., if subclass is not an instance of type), instead of raising TypeError.

Parameters

- subclass (type) – The target class to check.
- baseclass (type) – The base class subclass will be checked against.

```python
>>> class MyObject(object):
...     pass
... >>> issubclass(MyObject, object)  # always a fun fact
... True
>>> issubclass('hi', 'friend')
... False
```

boltons.typeutils.make_sentinel(name='_MISSING', var_name=None)

Creates and returns a new instance of a new class, suitable for usage as a “sentinel”, a kind of singleton often used to indicate a value is missing when None is a valid input.

Parameters

- name (str) – Name of the Sentinel
- var_name (str) – Set this name to the name of the variable in its respective module enable pickleability.

```python
>>> make_sentinel(var_name='_MISSING')
... _MISSING
```

The most common use cases here in boltons are as default values for optional function arguments, partly because of its less-confusing appearance in automatically generated documentation. Sentinels also function well as placeholders in queues and linked lists.

Note: By design, additional calls to make_sentinel with the same values will not produce equivalent objects.

```python
>>> make_sentinel('TEST') == make_sentinel('TEST')
... False
>>> type(make_sentinel('TEST')) == type(make_sentinel('TEST'))
... False
```

4.27 urlutils - Structured URL

urlutils is a module dedicated to one of software’s most versatile, well-aged, and beloved data structures: the URL, also known as the Uniform Resource Locator.
Among other things, this module is a full reimplementation of URLs, without any reliance on the `urlparse` or `urllib` standard library modules. The centerpiece and top-level interface of urlutils is the `URL` type. Also featured is the `find_all_links()` convenience function. Some low-level functions and constants are also below.

The implementations in this module are based heavily on RFC 3986 and RFC 3987, and incorporates details from several other RFCs and W3C documents.

New in version 17.2.

### 4.27.1 The URL type

```python
class boltons.urlutils.URL(url="")
```

The URL is one of the most ubiquitous data structures in the virtual and physical landscape. From blogs to billboards, URLs are so common, that it’s easy to overlook their complexity and power.

There are 8 parts of a URL, each with its own semantics and special characters:

- `scheme`
- `username`
- `password`
- `host`
- `port`
- `path`
- `query_params` (query string parameters)
- `fragment`

Each is exposed as an attribute on the URL object. RFC 3986 offers this brief structural summary of the main URL components:

```
foo://user:pass@example.com:8042/over/there?name=ferret#nose
\_/   \_______/ \_________/ \__/
  |    |    |    |    |    |    |
  scheme userinfo host port path query fragment
```

And here’s how that example can be manipulated with the URL type:

```python
>>> url = URL('foo://example.com:8042/over/there?name=ferret#nose')
>>> print(url.host)
example.com
>>> print(url.get_authority())
example.com:8042
>>> print(url.qp['name'])  # qp is a synonym for query_params
ferret
```

URL’s approach to encoding is that inputs are decoded as much as possible, and data remains in this decoded state until re-encoded using the `to_text()` method. In this way, it’s similar to Python’s current approach of encouraging immediate decoding of bytes to text.

Note that URL instances are mutable objects. If an immutable representation of the URL is desired, the string from `to_text()` may be used. For an immutable, but almost-as-featureful, URL object, check out the hyperlink package.

**scheme**

The scheme is an ASCII string, normally lowercase, which specifies the semantics for the rest of the URL, as well as network protocol in many cases. For example, “http” in “http://hatnote.com”.

---

**86 Chapter 4. Section listing**
**username**

The username is a string used by some schemes for authentication. For example, “public” in “ftp://public@example.com”.

**password**

The password is a string also used for authentication. Technically deprecated by RFC 3986 Section 7.5, they’re still used in cases when the URL is private or the password is public. For example “password” in “db://private:password@127.0.0.1”.

**host**

The host is a string used to resolve the network location of the resource, either empty, a domain, or IP address (v4 or v6). “example.com”, “127.0.0.1”, and “::1” are all good examples of host strings.

Per spec, fully-encoded output from `to_text()` is IDNA encoded for compatibility with DNS.

**port**

The port is an integer used, along with `host`, in connecting to network locations. 8080 is the port in “http://localhost:8080/index.html”.

**Note:** As is the case for 80 for HTTP and 22 for SSH, many schemes have default ports, and Section 3.2.3 of RFC 3986 states that when a URL’s port is the same as its scheme’s default port, the port should not be emitted:

```python
>>> url = URL(u'https://github.com:443/mahmoud/boltons').to_text()
u'https://github.com/mahmoud/boltons'
```

Custom schemes can register their port with `register_scheme()`. See `URL.default_port` for more info.

**path**

The string starting with the first leading slash after the authority part of the URL, ending with the first question mark. Often percent-quoted for network use. “/a/b/c” is the path of “http://example.com/a/b/c?d=e”.

**path_parts**

The tuple form of `path`, split on slashes. Empty slash segments are preserved, including that of the leading slash:

```python
>>> url = URL(u'http://example.com/a/b/c')
>>> url.path_parts
(u'', u'a', u'b', u'c')
```

**query_params**

An instance of `QueryParamDict`, an `OrderedMultiDict` subtype, mapping textual keys and values which follow the first question mark after the `path`. Also available as the handy alias `qp`:

```python
>>> url = URL('http://boltons.readthedocs.io/en/latest/?utm_source=docs&→ sphinx=ok')
>>> urlqp.keys()
[u'utm_source', u'sphinx']
```

Also percent-encoded for network use cases.

**fragment**

The string following the first ‘#’ after the `query_params` until the end of the URL. It has no inherent internal structure, and is percent-quoted.
classmethod from_parts(scheme=None, host=None, path_parts=(), query_params=(), fragment=u'', port=None, username=None, password=None)

Build a new URL from parts. Note that the respective arguments are not in the order they would appear in a URL:

Parameters

- **scheme** *(str)* – The scheme of a URL, e.g., ‘http’
- **host** *(str)* – The host string, e.g., ‘hatnote.com’
- **path_parts** *(tuple)* – The individual text segments of the path, e.g., (‘post’, ‘123’)
- **query_params** *(dict)* – An OMD, dict, or list of (key, value) pairs representing the keys and values of the URL’s query parameters.
- **fragment** *(str)* – The fragment of the URL, e.g., ‘anchor1’
- **port** *(int)* – The integer port of URL, automatic defaults are available for registered schemes.
- **username** *(str)* – The username for the userinfo part of the URL.
- **password** *(str)* – The password for the userinfo part of the URL.

Note that this method does relatively little validation. *URL.to_text()* should be used to check if any errors are produced while composing the final textual URL.

to_text(full_quote=False)

Render a string representing the current state of the URL object.

```python
>>> url = URL('http://listen.hatnote.com')
>>> url.fragment = 'en'
>>> print(url.to_text())
http://listen.hatnote.com#en
```

By setting the *full_quote* flag, the URL can either be fully quoted or minimally quoted. The most common characteristic of an encoded-URL is the presence of percent-encoded text (e.g., %60). Unquoted URLs are more readable and suitable for display, whereas fully-quoted URLs are more conservative and generally necessary for sending over the network.

default_port

Return the default port for the currently-set scheme. Returns *None* if the scheme is unrecognized. See *register_scheme()* above. If *port* matches this value, no port is emitted in the output of *to_text()*.

Applies the same ‘+’ heuristic detailed in *URL.uses_netloc()*.

uses_netloc

Whether or not a URL uses : or // to separate the scheme from the rest of the URL depends on the scheme’s own standard definition. There is no way to infer this behavior from other parts of the URL. A scheme either supports network locations or it does not.

The URL type’s approach to this is to check for explicitly registered schemes, with common schemes like HTTP preregistered. This is the same approach taken by *urlparse*.

URL adds two additional heuristics if the scheme as a whole is not registered. First, it attempts to check the subpart of the scheme after the last + character. This adds intuitive behavior for schemes like git+ssh. Second, if a URL with an unrecognized scheme is loaded, it will maintain the separator it sees.

```python
>>> print(URL('fakescheme://test.com').to_text())
fakescheme://test.com
```

(continues on next page)
boltons Documentation, Release 20.1.0

get_authority (full_quote=False, with_userinfo=False)
Used by URL schemes that have a network location, get_authority() combines username, password, host, and port into one string, the authority, that is used for connecting to a network-accessible resource.

Used internally by to_text() and can be useful for labeling connections.

```python
>>> url = URL('ftp://user@ftp.debian.org:2121/debian/README')
>>> print(url.get_authority())
ftp.debian.org:2121
```

Parameters

- **full_quote (bool)** – Whether or not to apply IDNA encoding. Defaults to False.
- **with_userinfo (bool)** – Whether or not to include username and password, technically part of the authority. Defaults to False.

normalize (with_case=True)
Resolve any “.” and “..” references in the path, as well as normalize scheme and host casing. To turn off case normalization, pass with_case=False.

More information can be found in Section 6.2.2 of RFC 3986.

navigate (dest)
Factory method that returns a _new_ URL based on a given destination, dest. Useful for navigating those relative links with ease.

The newly created URL is normalized before being returned.

```python
>>> url = URL('http://boltons.readthedocs.io')
>>> url.navigate('en/latest/')
URL(u'http://boltons.readthedocs.io/en/latest/')
```

Parameters **dest (str)** – A string or URL object representing the destination

More information can be found in Section 5 of RFC 3986.

Related functions

boltons.urlutils.find_all_links (text, with_text=False, default_scheme='https', schemes=())
This function uses heuristics to searches plain text for strings that look like URLs, returning a list of URL objects. It supports limiting the accepted schemes, and returning interleaved text as well.

```python
>>> find_all_links('Visit https://boltons.rtfd.org!')
[URL(u'https://boltons.rtfd.org')]  # URL objects
>>> find_all_links('Visit https://boltons.rtfd.org!', with_text=True)
[u'Visit ', URL(u'https://boltons.rtfd.org'), u'!']  # Interleaved text
```

Parameters
• **text** *(str)* – The text to search.
• **with_text** *(bool)* – Whether or not to interleave plaintext blocks with the returned URL objects. Having all tokens can be useful for transforming the text, e.g., replacing links with HTML equivalents. Defaults to False.
• **default_scheme** *(str)* – Many URLs are written without the scheme component. This function can match a reasonable subset of those, provided default_scheme is set to a string. Set to False to disable matching scheme-less URLs. Defaults to 'https'.
• **schemes** *(list)* – A list of strings that a URL’s scheme must match in order to be included in the results. Defaults to empty, which matches all schemes.

**Note:** Currently this function does not support finding IPv6 addresses or URLs with netloc-less schemes, like mailto.

```python
boltons.urlutils.register_scheme(text, uses_netloc=None, default_port=None)
```

Registers new scheme information, resulting in correct port and slash behavior from the URL object. There are dozens of standard schemes preregistered, so this function is mostly meant for proprietary internal customizations or stopgaps on missing standards information. If a scheme seems to be missing, please file an issue!

**Parameters**

- **text** *(str)* – Text representing the scheme. (the ‘http’ in ‘http://hatnote.com’)
- **uses_netloc** *(bool)* – Does the scheme support specifying a network host? For instance, “http” does, “mailto” does not.
- **default_port** *(int)* – The default port, if any, for netloc-using schemes.

### 4.27.2 Low-level functions

A slew of functions used internally by `URL`.

```python
boltons.urlutils.parse_url(url_text)
```

Used to parse the text for a single URL into a dictionary, used internally by the `URL` type.

Note that “URL” has a very narrow, standards-based definition. While `parse_url()` may raise `URLParseError` under a very limited number of conditions, such as non-integer port, a surprising number of strings are technically valid URLs. For instance, the text "url" is a valid URL, because it is a relative path.

In short, do not expect this function to validate form inputs or other more colloquial usages of URLs.

```python
>>> res = parse_url('http://127.0.0.1:3000/?a=1')
>>> sorted(res.keys())  # res is a basic dictionary
['_netloc_sep', 'authority', 'family', 'fragment', 'host', 'password', 'path', 'port', 'query', 'scheme', 'username']
```

```python
boltons.urlutils.parse_host(host)
```

Low-level function used to parse the host portion of a URL.

Returns a tuple of (family, host) where family is a `socket` module constant or None, and host is a string.

```python
>>> parse_host('googlewebsite.com') == (None, 'googlewebsite.com')
True
>>> parse_host('[:1]') == (socket.AF_INET6, '[:1]')
```

(continues on next page)
Odd doctest formatting above due to py3’s switch from `int` to `enums` for `socket` constants.

```python
>>> parse_host('192.168.1.1') == (socket.AF_INET, '192.168.1.1')
True
```

boltons.urlutils.parse_qsl(qs, keep_blank_values=True, encoding='utf8')
Converts a query string into a list of (key, value) pairs.

boltons.urlutils.resolve_pathParts(path_parts)
Normalize the URL path by resolving segments of `.` and `..`, resulting in a dot-free path. See RFC 3986 section 5.2.4, Remove Dot Segments.

```python
class boltons.urlutils.QueryParamDict(*args, **kwargs)
A subclass of `OrderedMultiDict` specialized for representing query string values. Everything is fully unquoted on load and all parsed keys and values are strings by default.

As the name suggests, multiple values are supported and insertion order is preserved.
```

```python
>>> qp = QueryParamDict.from_text(u'key=val1&key=val2&utm_source=rtd')
>>> qp.getlist('key')
[u'val1', u'val2']
>>> qp['key']
'val2'
>>> qp.add('key', 'val3')
>>> qp.to_text()  # 'key=val1&key=val2&utm_source=rtd&key=val3'
```

See `OrderedMultiDict` for more API features.

```python
classmethod from_text(query_string)
Parse `query_string` and return a new `QueryParamDict`.

to_text(full_quote=False)
Render and return a query string.

Parameters full_quote (bool) – Whether or not to percent-quote special characters or leave them decoded for readability.
```

Quoting

URLs have many parts, and almost as many individual “quoting” (encoding) strategies.

boltons.urlutils.quote_userinfoPart(text, full_quote=True)
Quote special characters in either the username or password section of the URL. Note that userinfo in URLs is considered deprecated in many circles (especially browsers), and support for percent-encoded userinfo can be spotty.

boltons.urlutils.quote_pathPart(text, full_quote=True)
Percent-encode a single segment of a URL path.

boltons.urlutils.quote_queryPart(text, full_quote=True)
Percent-encode a single query string key or value.

boltons.urlutils.quote_fragmentPart(text, full_quote=True)
Quote the fragment part of the URL. Fragments don’t have subdelimiters, so the whole URL fragment can be passed.

There is however, only one unquoting strategy:
boltons.urlutils.unquote(string, encoding='utf-8', errors='replace')

Percent-decode a string, by replacing %xx escapes with their single-character equivalent. The optional encoding and errors parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the bytes.decode() method. By default, percent-encoded sequences are decoded with UTF-8, and invalid sequences are replaced by a placeholder character.

```python
>>> unquote(u'abc%20def')
'abc def'
```

### 4.27.3 Useful constants

boltons.urlutils.SCHEME_PORT_MAP

A mapping of URL schemes to their protocols' default ports. Painstakingly assembled from the IANA scheme registry, port registry, and independent research.

Keys are lowercase strings, values are integers or None, with None indicating that the scheme does not have a default port (or may not support ports at all):

```python
>>> boltons.urlutils.SCHEME_PORT_MAP['http']
80
>>> boltons.urlutils.SCHEME_PORT_MAP['file']
None
```

See URL.port for more info on how it is used. See NO_NETLOC_SCHEMES for more scheme info.

Also available in JSON.

boltons.urlutils.NO_NETLOC_SCHEMES

This is a set of schemes explicitly do not support network resolution, such as “mailto” and “urn”.

(For a quick reference you can ctrl-F, see the genindex.)
boltons.cacheutils, 10
boltons.debugutils, 15
boltons.dictutils, 16
boltons.ecoutils, 22
boltons.fileutils, 24
boltons.formatutils, 27
boltons.funcutils, 29
boltons.gcutils, 34
boltons.ioutils, 35
boltons.iterutils, 37
boltons.jsonutils, 47
boltons.listutils, 48
boltons.mathutils, 49
boltons.mboxutils, 51
boltons.namedutils, 51
boltons.pathutils, 52
boltons.queueutils, 54
boltons.setutils, 55
boltons.socketutils, 58
boltons.statsutils, 62
boltons.strutils, 69
boltons.tableutils, 74
boltons.tbutils, 76
boltons.timeoututils, 80
boltons.typeutils, 84
boltons.urlutils, 85
Index

A
a10n() (in module boltons.strutils), 72
add() (boltons.cacheutils ThresholdCounter method), 14
add() (boltons.dictutils ManyToMany method), 21
add() (boltons.dictutils OrderedMultiDict method), 17
add() (boltons.queueutils BasePriorityQueue method), 55
add_arg() (boltons.funcutils FunctionBuilder method), 31
addlist() (boltons.dictutils OrderedMultiDict method), 17
append() (boltons.listutils BarrelList method), 48
args2cmd() (in module boltons.strutils), 73
args2sh() (in module boltons.strutils), 73
as_bin() (boltons.mathutils Bits method), 49
as_bytes() (boltons.mathutils Bits method), 49
as_int() (boltons.mathutils Bits method), 49
as_list() (boltons.mathutils Bits method), 49
asciify() (in module boltons.mathutils), 26
atomic_rename() (in module boltons.fileutils), 26
atomic_save() (in module boltons.fileutils), 25
AtomicSaver (class in boltons.fileutils), 25
augpath() (in module boltons.pathutils), 52

boltons.fileutils (module), 24
boltons.formatutils (module), 27
boltons.funcutils (module), 29
boltons.gcutils (module), 34
boltons.ioutils (module), 35
boltons.iterutils (module), 37
boltons.jsonutils (module), 47
boltons.listutils (module), 48
boltons.mathutils (module), 49
boltons.mboxutils (module), 51
boltons.namedutils (module), 51
boltons.pathutils (module), 52
boltons.queueutils (module), 54
boltons.setutils (module), 55
boltons.socketutils (module), 58
boltons.statsutils (module), 62
boltons.strutils (module), 69
boltons.tableutils (module), 74
boltons.tbuilts (module), 76
boltons.timeutils (module), 80
boltons.typeutils (module), 84
boltons.urlutils (module), 85
bucketize() (in module boltons.iterutils), 44
buffer() (boltons.socketutils BufferedSocket method), 59
BufferedSocket (class in boltons.socketutils), 58
bytes2human() (in module boltons.strutils), 71

B
backoff() (in module boltons.iterutils), 42
backoff_iter() (in module boltons.iterutils), 42
BarrelList (class in boltons.listutils), 48
BaseFormatField (class in boltons.formatutils), 28
BasePriorityQueue (class in boltons.queueutils), 55
Bits (class in boltons.mathutils), 49
BList (class in boltons.listutils), 48
boltons.cacheutils (module), 10
boltons.debugutils (module), 15
boltons.dictutils (module), 16
boltons.ecoutils (module), 22

C
cached() (in module boltons.cacheutils), 12
CachedInstancePartial (class in boltons.funcutils), 32
cachedmethod() (in module boltons.cacheutils), 13
cachedproperty() (in module boltons.cacheutils), 13
Callpoint (class in boltons.tbuilts), 78
callpoint_type (boltons.tbuilts ContextualTracebackInfo attribute), 79
callpoint_type (boltons.tbuilts TracebackInfo attribute), 77
camel2under () (in module boltons.strutils), 69
cardinalize() (in module boltons.strutils), 70
cell() (in module boltons.mathutils), 50
Central (boltons.timeutils attribute), 84
chunked() (in module boltons.iterutils), 38
chunked_iter() (in module boltons.iterutils), 38
clamp() (in module boltons.mathutils), 50
classproperty (class in boltons.typeutils), 84
clear() (boltons.cacheutils.LRI method), 11
clear() (boltons.dictutils.FrozenDict method), 22
clear() (boltons.dictutils.OneToOne method), 20
clear() (boltons.dictutils.OrderedMultiDict method), 17
clear() (boltons.setutils.IndexedSet method), 56
clear_cache() (boltons.statsutils.Stats method), 64
close() (boltons.socketutils.BufferedSocket method), 59
complement() (in module boltons.setutils), 57
ConnectionClosed, 62
ConstantTZInfo (class in boltons.timeutils), 84
construct_format_field_str() (in module boltons.formatutils), 28
ContextualCallpoint (class in boltons.tbutils), 79
ContextualExceptionInfo (class in boltons.tbutils), 79
ContextualTracebackInfo (class in boltons.tbutils), 79
copy() (boltons.cacheutils.LRI method), 11
copy() (boltons.dictutils.OneToOne method), 20
copy() (boltons.dictutils.OrderedMultiDict method), 17
copy_function() (in module boltons.funcutils), 32
copytree() (in module boltons.fileutils), 25
count (boltons.statsutils.Stats attribute), 64
count() (boltons.listutils.BarrelList method), 48
count() (boltons.setutils.IndexedSet method), 56
counts() (boltons.dictutils.OrderedMultiDict method), 17
cur_byte_pos (boltons.jsonutils.JSONLIterator attribute), 47
date() (boltons.setutils.IndexedSet method), 56
date_to_timestamp() (in module boltons.timeutils), 82
dummyfile (class in boltons.fileutils), 27
E
Eastern (boltons.timeutils attribute), 84
elements() (boltons.cacheutils.ThresholdCounter method), 14
Error, 61
escape_shell_args() (in module boltons.strutils), 73
ExceptionInfo (class in boltons.tbutils), 77
expandpath() (in module boltons.pathutils), 54
extend() (boltons.listutils.BarrelList method), 49
extend() (boltons.tableutils.Table method), 75
F
family (boltons.socketutils.BufferedSocket attribute), 59
fileno() (boltons.socketutils.BufferedSocket method), 59
FilePerms (class in boltons.fileutils), 26
find_all() (in module boltons.urlutils), 89
find_hashtags() (in module boltons.strutils), 72
first() (in module boltons.iterutils), 46
flatten() (in module boltons.iterutils), 42
flatten_iter() (in module boltons.iterutils), 42
floor() (in module boltons.mathutils), 50
flush() (boltons.mboxutils.mbox_readonlydir method), 51
flush() (boltons.socketutils.BufferedSocket method), 59
format_exp_repr() (in module boltons.funcutils), 32
format_histogram() (boltons.statsutils.Stats method), 64
format_histogram_counts() (in module boltons.statsutils), 67
format_int_list() (in module boltons.statsutils), 74
format_invocation() (in module boltons.statsutils), 32
format_nonexp_repr() (in module boltons.statsutils), 33
fragment (boltons.urlutils.URL attribute), 87
frange() (in module boltons.iterutils), 43
from_bin() (boltons.mathutils.Bits class method), 49
from_bytes() (boltons.mathutils.Bits class method), 49
from_current() (boltons.tbutils.Callpoint class method), 78
from_current() (boltons.tbutils.ExceptionInfo class method), 77
<table>
<thead>
<tr>
<th>Method</th>
<th>Class/Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>setdefault()</td>
<td>boltons.dictutils.OrderedMultiDict method</td>
</tr>
<tr>
<td>setmaxsize()</td>
<td>boltons.socketutils.BufferedSocket method</td>
</tr>
<tr>
<td>setsockopt()</td>
<td>boltons.socketutils.BufferedSocket method</td>
</tr>
<tr>
<td>settimeout()</td>
<td>boltons.socketutils.BufferedSocket method</td>
</tr>
<tr>
<td>shutdown()</td>
<td>boltons.socketutils.BufferedSocket method</td>
</tr>
<tr>
<td>shrinkuser()</td>
<td>in module boltons.pathutils</td>
</tr>
<tr>
<td>singularize()</td>
<td>in module boltons.strutils</td>
</tr>
<tr>
<td>skewness(boltons.statsutils.Stats attribute)</td>
<td></td>
</tr>
<tr>
<td>skewness()</td>
<td>in module boltons.statsutils</td>
</tr>
<tr>
<td>slugify()</td>
<td>in module boltons.strutils</td>
</tr>
<tr>
<td>soft_sorted()</td>
<td>in module boltons.iterutils</td>
</tr>
<tr>
<td>sort(boltons.listutils.BarrelList method)</td>
<td></td>
</tr>
<tr>
<td>sort()</td>
<td>boltons.setutils.IndexedSet method</td>
</tr>
<tr>
<td>sorted()</td>
<td>boltons.dictutils.OrderedMultiDict method</td>
</tr>
<tr>
<td>SortedPriorityQueue</td>
<td>(class in boltons.queueutils)</td>
</tr>
<tr>
<td>sortedvalues()</td>
<td>(boltons.dictutils.OrderedMultiDict method)</td>
</tr>
<tr>
<td>source_file</td>
<td>(boltons.tbutils.ParsedException attribute)</td>
</tr>
<tr>
<td>split()</td>
<td>in module boltons.iterutils</td>
</tr>
<tr>
<td>split_iter()</td>
<td>in module boltons.iterutils</td>
</tr>
<tr>
<td>split_punct_ws()</td>
<td>in module boltons.iterutils</td>
</tr>
<tr>
<td>SpooledBytesIO</td>
<td>(class in boltons.ioutils)</td>
</tr>
<tr>
<td>SpooledStringIO</td>
<td>(class in boltons.ioutils)</td>
</tr>
<tr>
<td>Stats</td>
<td>(class in boltons.statsutils)</td>
</tr>
<tr>
<td>std_dev(boltons.statsutils.Stats attribute)</td>
<td></td>
</tr>
<tr>
<td>std_dev()</td>
<td>in module boltons.statsutils</td>
</tr>
<tr>
<td>strip_ansi()</td>
<td>in module boltons.statsutils</td>
</tr>
<tr>
<td>strpdate()</td>
<td>in module boltons.timeutils</td>
</tr>
<tr>
<td>subdict()</td>
<td>in module boltons.dictutils</td>
</tr>
<tr>
<td>symmetric_difference()</td>
<td>(boltons.setutils.IndexedSet method)</td>
</tr>
<tr>
<td>symmetric_difference_update()</td>
<td>(boltons.setutils.IndexedSet method)</td>
</tr>
<tr>
<td>Table</td>
<td>(class in boltons.tableutils)</td>
</tr>
<tr>
<td>to_dict()</td>
<td>(boltons.tbutils.ContextualCallpoint method)</td>
</tr>
<tr>
<td>to_dict()</td>
<td>(boltons.tbutils.ExceptionInfo method)</td>
</tr>
<tr>
<td>to_dict()</td>
<td>(boltons.tbutils.ParsedException method)</td>
</tr>
<tr>
<td>to_dict()</td>
<td>(boltons.tbutils.TracebackInfo method)</td>
</tr>
<tr>
<td>to_html()</td>
<td>(boltons.tableutils.Table method)</td>
</tr>
<tr>
<td>to_string()</td>
<td>(boltons.tbutils.ParsedException method)</td>
</tr>
<tr>
<td>to_text()</td>
<td>(boltons.tableutils.Table method)</td>
</tr>
<tr>
<td>to_text()</td>
<td>(boltons.umlutils.QueryParamDict method)</td>
</tr>
<tr>
<td>to_text()</td>
<td>(boltons.umlutils.URL method)</td>
</tr>
<tr>
<td>toggle_gc</td>
<td>(boltons.dictutils.OrderedMultiDict method)</td>
</tr>
<tr>
<td>toggle_gc_postcollect</td>
<td>in module boltons.gcutils</td>
</tr>
<tr>
<td>tokenize_format_str</td>
<td>(in module boltons.formattools)</td>
</tr>
<tr>
<td>total_seconds()</td>
<td>(in module boltons.timeutils)</td>
</tr>
<tr>
<td>TracebackInfo</td>
<td>(class in boltons.tbutils)</td>
</tr>
<tr>
<td>trim_relative()</td>
<td>(boltons.statsutils.Stats method)</td>
</tr>
<tr>
<td>trimean</td>
<td>(boltons.statsutils.Stats attribute)</td>
</tr>
<tr>
<td>trimean()</td>
<td>(in module boltons.statsutils)</td>
</tr>
<tr>
<td>type(boltons.socketutils.BufferedSocket attribute)</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>under2camel()</td>
<td>(in module boltons.strutils)</td>
</tr>
<tr>
<td>union(boltons.setutils.IndexedSet method)</td>
<td></td>
</tr>
<tr>
<td>unique(boltons.dictutils.OneToOne class method)</td>
<td></td>
</tr>
<tr>
<td>unique()</td>
<td>(in module boltons.iterutils)</td>
</tr>
<tr>
<td>unique_iter()</td>
<td>(in module boltons.iterutils)</td>
</tr>
<tr>
<td>unit_len()</td>
<td>(in module boltons.statsutils)</td>
</tr>
<tr>
<td>unquote()</td>
<td>(in module boltons.umlutils)</td>
</tr>
<tr>
<td>unwrap_text()</td>
<td>(in module boltons.statsutils)</td>
</tr>
<tr>
<td>update(boltons.cacheutils.LRI method)</td>
<td></td>
</tr>
<tr>
<td>update()</td>
<td>(boltons.cacheutils.ThresholdCounter method)</td>
</tr>
<tr>
<td>update()</td>
<td>(boltons.dictutils.FrozenDict method)</td>
</tr>
<tr>
<td>update()</td>
<td>(boltons.dictutils.ManyToMany method)</td>
</tr>
<tr>
<td>update()</td>
<td>(boltons.dictutils.OneToMany method)</td>
</tr>
<tr>
<td>update()</td>
<td>(boltons.dictutils.OrderedMultiDict method)</td>
</tr>
<tr>
<td>update()</td>
<td>(boltons.setutils.IndexedSet method)</td>
</tr>
<tr>
<td>update_extend()</td>
<td>(boltons.dictutils.OrderedMultiDict method)</td>
</tr>
<tr>
<td>updated()</td>
<td>(boltons.dictutils.FrozenDict method)</td>
</tr>
<tr>
<td>URL</td>
<td>(class in boltons.umlutils)</td>
</tr>
<tr>
<td>username(boltons.umlutils.URL attribute)</td>
<td></td>
</tr>
<tr>
<td>uses_netloc(boltons.umlutils.URL attribute)</td>
<td></td>
</tr>
<tr>
<td>USTimeZone</td>
<td>(class in boltons.timeutils)</td>
</tr>
</tbody>
</table>
UTC (boltons.timeutils attribute), 83

V

val (boltons.mathutils.Bits attribute), 49
values() (boltons.dictutils.OrderedMultiDict method), 20
variance (boltons.statsutils.Stats attribute), 66
variance() (in module boltons.statsutils), 68
viewitems() (boltons.dictutils.OrderedMultiDict method), 20
viewkeys() (boltons.dictutils.OrderedMultiDict method), 20
viewvalues() (boltons.dictutils.OrderedMultiDict method), 20

W

windowed() (in module boltons.iterutils), 39
windowed_iter() (in module boltons.iterutils), 39
wrap_trace() (in module boltons.debugutils), 15
wraps() (in module boltons.funcutils), 29

X

xfrange() (in module boltons.iterutils), 43