
pydash Documentation

Release 4.0.3

Derrick Gilland

Apr 20, 2017

Contents

1	Links	3
2	Quickstart	5
3	Guide	7
3.1	Installation	7
3.2	Quickstart	7
3.3	Lodash Differences	8
3.4	Callbacks	9
3.5	Deep Path Strings	11
3.6	Method Chaining	11
3.7	Templating	13
3.8	Upgrading	14
4	API Reference	21
4.1	API Reference	21
5	Project Info	161
5.1	License	161
5.2	Versioning	161
5.3	Changelog	162
5.4	Authors	183
5.5	Contributing	184
5.6	Kudos	185
6	Indices and Tables	187
	Python Module Index	189

The kitchen sink of Python utility libraries for doing “stuff” in a functional way. Based on the [Lo-Dash](#) Javascript library.

CHAPTER 1

Links

- Project: <https://github.com/dgilland/pydash>
- Documentation: <http://pydash.readthedocs.org>
- PyPi: <https://pypi.python.org/pypi/pydash/>
- TravisCI: <https://travis-ci.org/dgilland/pydash>

CHAPTER 2

Quickstart

The functions available from pydash can be used in two styles.

The first is by using the module directly or importing from it:

```
>>> import pydash
>>> from pydash import flatten

# Arrays
>>> flatten([1, 2, [3, [4, 5, [6, 7]]]])
[1, 2, 3, [4, 5, [6, 7]]]

>>> pydash.flatten_deep([1, 2, [3, [4, 5, [6, 7]]]])
[1, 2, 3, 4, 5, 6, 7]

# Collections
>>> pydash.map_([{'name': 'moe', 'age': 40}, {'name': 'larry', 'age': 50}], 'name')
['moe', 'larry']

# Functions
>>> curried = pydash.curry(lambda a, b, c: a + b + c)
>>> curried(1, 2)(3)
6

# Objects
>>> pydash.omit({'name': 'moe', 'age': 40}, 'age')
{'name': 'moe'}

# Utilities
>>> pydash.times(3, lambda index: index)
[0, 1, 2]

# Chaining
>>> pydash.chain([1, 2, 3, 4]).without(2, 3).reject(lambda x: x > 1).value()
[1]
```

The second style is to use the `py_` or `_` instances (they are the same object as two different aliases):

```
>>> from pydash import py_

# Method calling which is equivalent to pydash.flatten(...)
>>> py_.flatten([1, 2, [3, [4, 5, [6, 7]]]])
[1, 2, 3, [4, 5, [6, 7]]]

# Method chaining which is equivalent to pydash.chain(...)
>>> py_([1, 2, 3, 4]).without(2, 3).reject(lambda x: x > 1).value()
[1]

# Late method chaining
>>> py_().without(2, 3).reject(lambda x: x > 1)([1, 2, 3, 4])
[1]
```

See also:

For further details consult [API Reference](#).

Installation

pydash requires Python ≥ 2.6 or ≥ 3.3 . It has no external dependencies.

To install from **PyPi**:

```
pip install pydash
```

Quickstart

The functions available from **pydash** can be used in two styles.

The first is by using the module directly or importing from it:

```
>>> import pydash
>>> from pydash import flatten

# Arrays
>>> flatten([1, 2, [3, [4, 5, [6, 7]]]])
[1, 2, 3, [4, 5, [6, 7]]]

>>> pydash.flatten_deep([1, 2, [3, [4, 5, [6, 7]]]])
[1, 2, 3, 4, 5, 6, 7]

# Collections
>>> pydash.map_([{'name': 'moe', 'age': 40}, {'name': 'larry', 'age': 50}], 'name')
['moe', 'larry']

# Functions
>>> curried = pydash.curry(lambda a, b, c: a + b + c)
>>> curried(1, 2)(3)
6
```

```
# Objects
>>> pydash.omit({'name': 'moe', 'age': 40}, 'age')
{'name': 'moe'}

# Utilities
>>> pydash.times(3, lambda index: index)
[0, 1, 2]

# Chaining
>>> pydash.chain([1, 2, 3, 4]).without(2, 3).reject(lambda x: x > 1).value()
[1]
```

The second style is to use the `py_` or `_` instances (they are the same object as two different aliases):

```
>>> from pydash import py_

# Method calling which is equivalent to pydash.flatten(...)
>>> py_.flatten([1, 2, [3, [4, 5, [6, 7]]]])
[1, 2, 3, [4, 5, [6, 7]]]

# Method chaining which is equivalent to pydash.chain(...)
>>> py_([1, 2, 3, 4]).without(2, 3).reject(lambda x: x > 1).value()
[1]

# Late method chaining
>>> py_().without(2, 3).reject(lambda x: x > 1)([1, 2, 3, 4])
[1]
```

See also:

For further details consult [API Reference](#).

Lodash Differences

Naming Conventions

pydash adheres to the following conventions:

- Function names use `snake_case` instead of `camelCase`.
- Any Lodash function that shares its name with a reserved Python keyword will have an `_` appended after it (e.g. `filter` in Lodash would be `filter_` in pydash).
- Lodash's `toArray()` is pydash's `to_list()`.
- Lodash's `functions()` is pydash's `callable()`. This particular name difference was chosen in order to allow for the `functions.py` module file to exist at root of the project. Previously, `functions.py` existed in `pydash/api/` but in `v2.0.0`, it was decided to move everything in `api/` to `pydash/`. Therefore, to avoid import ambiguities, the `functions()` function was renamed.
- Lodash's `is_native()` is pydash's `is_builtin()`. This aligns better with Python's builtins terminology.

Callbacks

There are a few differences between extra callback style support:

- Pydash has an explicit shallow property access of the form `['some_property']` as in `pydash.map_([{'a.b': 1, 'a': {'b': 3}}, {'a.b': 2, 'a': {'b': 4}}], ['a.b'])` would evaluate to `[1, 2]` and not `[3, 4]` (as would be the case for `'a.b'`).

Extra Functions

In addition to porting Lodash, pydash contains functions found in [lodashcontrib](#), [lodashdeep](#), [lodashmath](#), and [underscorestring](#).

Function Behavior

Some of pydash's functions behave differently:

- `pydash.utilities.memoize()` uses all passed in arguments as the cache key by default instead of only using the first argument.

Templating

- pydash doesn't have `template()`. See [Templating](#) for more details.

Callbacks

For functions that support callbacks, there are several callback styles that can be used.

Callable Style

The most straight-forward callback is a regular callable object. For pydash functions that pass multiple arguments to their callback, the callable's argument signature does not need to support all arguments. Pydash's callback system will try to infer the number of supported arguments of the callable and only pass those arguments to the callback. However, there may be some edge cases where this will fail in which case one will need to wrap the callable in a `lambda` or `def ...` style function.

The arguments passed to most callbacks are:

```
callback(item, index, obj)
```

where `item` is an element of `obj`, `index` is the dict or list index, and `obj` is the original object being passed in. But not all callbacks support these arguments. Some functions support fewer callback arguments. See [API Reference](#) for more details.

```
>>> users = [
...     {'name': 'Michelangelo', 'active': False},
...     {'name': 'Donatello', 'active': False},
...     {'name': 'Leonardo', 'active': True}
... ]

# Single argument callback.
>>> callback = lambda item: item['name'] == 'Donatello'
>>> pydash.find_index(users, callback)
1
```

```
# Two argument callback.
>>> callback = lambda item, index: index == 3
>>> pydash.find_index(users, callback)
-1

# Three argument callback.
>>> callback = lambda item, index, obj: obj[index]['active']
>>> pydash.find_index(users, callback)
2
```

Shallow Property Style

The shallow property style callback is specified as a one item list containing the property value to return from an element. Internally, `pydash.utilities.prop()` is used to create the callback.

```
>>> users = [
...     {'name': 'Michelangelo', 'active': False},
...     {'name': 'Donatello', 'active': False},
...     {'name': 'Leonardo', 'active': True}
... ]
>>> pydash.find_index(users, ['active'])
2
```

Deep Property Style

The deep property style callback is specified as a deep property string of the nested object value to return from an element. Internally, `pydash.utilities.deep_prop()` is used to create the callback. See [Deep Path Strings](#) for more details.

```
>>> users = [
...     {'name': 'Michelangelo', 'location': {'city': 'Rome'}},
...     {'name': 'Donatello', 'location': {'city': 'Florence'}},
...     {'name': 'Leonardo', 'location': {'city': 'Amboise'}}
... ]
>>> pydash.map_(users, 'location.city')
['Rome', 'Florence', 'Amboise']
```

Matches Property Style

The matches property style callback is specified as a two item list containing a property key and value and returns True when an element's key is equal to value, else False. Internally, `pydash.utilities.matches_property()` is used to create the callback.

```
>>> users = [
...     {'name': 'Michelangelo', 'active': False},
...     {'name': 'Donatello', 'active': False},
...     {'name': 'Leonardo', 'active': True}
... ]
>>> pydash.find_index(users, ['active', False])
0
>>> pydash.find_last_index(users, ['active', False])
1
```

Matches Style

The matches style callback is specified as a dict object and returns True when an element matches the properties of the object, else False. Internally, `pydash.utilities.matches()` is used to create the callback.

```
>>> users = [
...     {'name': 'Michelangelo', 'location': {'city': 'Rome'}},
...     {'name': 'Donatello', 'location': {'city': 'Florence'}},
...     {'name': 'Leonardo', 'location': {'city': 'Amboise'}}
... ]
>>> pydash.map_(users, {'location': {'city': 'Florence'}})
[False, True, False]
```

Deep Path Strings

A deep path string is used to access a nested data structure of arbitrary length. Each level is separated by a "." and can be used on both dictionaries and lists. If a "." is contained in one of the dictionary keys, then it can be escaped using "\". For accessing a dictionary key that is a number, it can be wrapped in brackets like "[1]".

Examples:

```
>>> data = {'a': {'b': {'c': [0, 0, {'d': [0, {1: 2}]}]}}}
>>> pydash.get(data, 'a.b.c.2.d.1.[1]')
2

>>> data = {'a': {'b.c.d': 2}}
>>> pydash.get(data, r'a.b\.c\.d')
2
```

Pydash's callback system supports the deep property style callback using deep path strings.

Method Chaining

Method chaining in pydash is quite simple.

An initial value is provided:

```
from pydash import py_
py_([1, 2, 3, 4])

# Or through the chain() function
import pydash
pydash.chain([1, 2, 3, 4])
```

Methods are chained:

```
py_([1, 2, 3, 4]).without(2, 3).reject(lambda x: x > 1)
```

A final value is computed:

```
result = py_([1, 2, 3, 4]).without(2, 3).reject(lambda x: x > 1).value()
```

Lazy Evaluation

Method chaining is deferred (lazy) until `.value()` is called:

```
>>> from __future__ import print_function
>>> from pydash import py_

>>> def echo(value): print(value)

>>> lazy = py_([1, 2, 3, 4]).for_each(echo)

# None of the methods have been called yet.

>>> result = lazy.value()
1
2
3
4

# Each of the chained methods have now been called.

>>> assert result == [1, 2, 3, 4]

>>> result = lazy.value()
1
2
3
4
```

Committing a Chain

If one wishes to create a new chain object seeded with the computed value of another chain, then one can use the `commit` method:

```
>>> committed = lazy.commit()
1
2
3
4

>>> committed.value()
[1, 2, 3, 4]

>>> lazy.value()
1
2
3
4
[1, 2, 3, 4]
```

Committing is equivalent to:

```
committed = py_(lazy.value())
```


Late Value Passing

In *v3.0.0* the concept of late value passing was introduced to method chaining. This allows method chains to be re-used with different root values supplied. Essentially, ad-hoc functions can be created via the chaining syntax.

```
>>> square_sum = py_().power(2).sum()
>>> assert square_sum([1, 2, 3]) == 14
>>> assert square_sum([4, 5, 6]) == 77

>>> square_sum_square = square_sum.power(2)
>>> assert square_sum_square([1, 2, 3]) == 196
>>> assert square_sum_square([4, 5, 6]) == 5929
```

Planting a Value

To replace the initial value of a chain, use the `plant` method which will return a cloned chained using the new initial value:

```
>>> chained = py_([1, 2, 3, 4]).power(2).sum()
>>> chained.value()
30
>>> rechained = chained.plant([5, 6, 7, 8])
>>> rechained.value()
174
>>> chained.value()
30
```

Module Access

Another feature of the `py_` object, is that it provides module access to `pydash`:

```
>>> import pydash
>>> from pydash import py_

>>> assert py_.add is pydash.add
>>> py_.add(1, 2) == pydash.add(1, 2)
True
```

Through `py_` any function that ends with `"_"` can be accessed without the trailing `"_"`:

```
>>> py_.filter([1, 2, 3], lambda x: x > 1) == pydash.filter_([1, 2, 3], lambda x: x >
↪ 1)
True
```

Templating

Templating has been purposely left out of `pydash`. Having a custom templating engine was never a goal of `pydash` even though `Lodash` includes one. There already exist many mature and battle-tested templating engines like `Jinja2` and `Mako` which are better suited to handling templating needs. However, if there was ever a strong request/justification for having templating in `pydash` (or a pull-request implementing it), then this decision could be re-evaluated.

Upgrading

From v3.x.x to v4.0.0

Start by reading the full list of changes in v4.0.0 at the [Changelog](#). There are a significant amount of backwards-incompatibilities that will likely need to be addressed:

- All function aliases have been removed in favor of having a single named function for everything. This was done to make things less confusing by having only a single named function that performs an action vs. potentially using two different names for the same function.
- A few functions have been removed whose functionality was duplicated by another function.
- Some functions have been renamed for consistency and to align with Lodash.
- Many functions have had their callback argument moved to another function to align with Lodash.
- The generic `callback` argument has been renamed to either `iteratee`, `predicate`, or `comparator`. This was done to make it clearer what the callback is doing and to align more with Lodash's naming conventions.

Once the shock of those backwards-incompatibilities has worn off, discover 72 new functions:

- 19 new array methods
 - `pydash.arrays.difference_by()`
 - `pydash.arrays.difference_with()`
 - `pydash.arrays.from_pairs()`
 - `pydash.arrays.intersection_by()`
 - `pydash.arrays.intersection_with()`
 - `pydash.arrays.nth()`
 - `pydash.arrays.pull_all()`
 - `pydash.arrays.sorted_index_by()`
 - `pydash.arrays.sorted_index_of()`
 - `pydash.arrays.sorted_last_index_by()`
 - `pydash.arrays.sorted_last_index_of()`
 - `pydash.arrays.sorted_uniq()`
 - `pydash.arrays.union_by()`
 - `pydash.arrays.union_with()`
 - `pydash.arrays.uniq_by()`
 - `pydash.arrays.uniq_with()`
 - `pydash.arrays.xor_by()`
 - `pydash.arrays.xor_with()`
 - `pydash.arrays.zip_object_deep()`
- 6 new collection methods
 - `pydash.collections.flat_map()`
 - `pydash.collections.flat_map_deep()`

- `pydash.collections.flat_depth()`
- `pydash.collections.flatten_depth()`
- `pydash.collections.invoke_map()`
- `pydash.collections.sample_size()`
- 2 new function methods
 - `pydash.functions.flip()`
 - `pydash.functions.unary()`
- 12 new object methods
 - `pydash.objects.assign_with()`
 - `pydash.objects.clone_deep_with()`
 - `pydash.objects.clone_with()`
 - `pydash.objects.invert_by()`
 - `pydash.objects.merge_with()`
 - `pydash.objects.omit_by()`
 - `pydash.objects.pick_by()`
 - `pydash.objects.set_with()`
 - `pydash.objects.to_integer()`
 - `pydash.objects.unset()`
 - `pydash.objects.update()`
 - `pydash.objects.udpate_with()`
- 8 new numerical methods
 - `pydash.numerical.clamp()`
 - `pydash.numerical.divide()`
 - `pydash.numerical.max_by()`
 - `pydash.numerical.mean_by()`
 - `pydash.numerical.min_by()`
 - `pydash.numerical.multiply()`
 - `pydash.numerical.subtract()`
 - `pydash.numerical.sum_by()`
- 4 new predicate methods
 - `pydash.predicates.eq()`
 - `pydash.predicates.is_equal_with()`
 - `pydash.predicates.is_match_with()`
 - `pydash.predicates.is_set()`
- 6 new string methods
 - `pydash.strings.lower_case()`

- `pydash.strings.lower_first()`
- `pydash.strings.to_lower()`
- `pydash.strings.to_upper()`
- `pydash.strings.upper_case()`
- `pydash.strings.upper_first()`

- 15 new utility methods

- `pydash.utilities.cond()`
- `pydash.utilities.conforms()`
- `pydash.utilities.conforms_to()`
- `pydash.utilities.default_to()`
- `pydash.utilities.nth_arg()`
- `pydash.utilities.over()`
- `pydash.utilities.over_every()`
- `pydash.utilities.over_some()`
- `pydash.utilities.range_right()`
- `pydash.utilities.stub_list()`
- `pydash.utilities.stub_dict()`
- `pydash.utilities.stub_false()`
- `pydash.utilities.stub_string()`
- `pydash.utilities.stub_true()`
- `pydash.utilities.to_path()`

From v2.x.x to v3.0.0

There were several breaking changes in v3.0.0:

- Make `to_string` convert `None` to empty string. (**breaking change**)
- Make the following functions work with empty strings and `None`: (**breaking change**)
 - `camel_case`
 - `capitalize`
 - `chars`
 - `chop`
 - `chop_right`
 - `class_case`
 - `clean`
 - `count_substr`
 - `decapitalize`
 - `ends_with`

- join
 - js_replace
 - kebab_case
 - lines
 - quote
 - re_replace
 - replace
 - series_phrase
 - series_phrase_serial
 - starts_with
 - surround
- Reorder function arguments for after from (n, func) to (func, n). **(breaking change)**
 - Reorder function arguments for before from (n, func) to (func, n). **(breaking change)**
 - Reorder function arguments for times from (n, callback) to (callback, n). **(breaking change)**
 - Reorder function arguments for js_match from (reg_exp, text) to (text, reg_exp). **(breaking change)**
 - Reorder function arguments for js_replace from (reg_exp, text, repl) to (text, reg_exp, repl). **(breaking change)**

And some potential breaking changes:

- Move arrays.join to strings.join **(possible breaking change)**.
- Rename join/implode's second parameter from delimiter to separator. **(possible breaking change)**
- Rename split/explode's second parameter from delimiter to separator. **(possible breaking change)**

Some notable new features/functions:

- 31 new string methods
 - `pydash.strings.chars()`
 - `pydash.strings.chop()`
 - `pydash.strings.chop_right()`
 - `pydash.strings.class_case()`
 - `pydash.strings.clean()`
 - `pydash.strings.count_substr()`
 - `pydash.strings.decapitalize()`
 - `pydash.strings.has_substr()`
 - `pydash.strings.human_case()`
 - `pydash.strings.insert_substr()`
 - `pydash.strings.lines()`
 - `pydash.strings.number_format()`

- `pydash.strings.pascal_case()`
- `pydash.strings.predecessor()`
- `pydash.strings.prune()`
- `pydash.strings.re_replace()`
- `pydash.strings.replace()`
- `pydash.strings.separator_case()`
- `pydash.strings.series_phrase()`
- `pydash.strings.series_phrase_serial()`
- `pydash.strings.slugify()`
- `pydash.strings.split()`
- `pydash.strings.strip_tags()`
- `pydash.strings.substr_left()`
- `pydash.strings.substr_left_end()`
- `pydash.strings.substr_right()`
- `pydash.strings.substr_right_end()`
- `pydash.strings.successor()`
- `pydash.strings.swap_case()`
- `pydash.strings.title_case()`
- `pydash.strings.unquote()`

- 1 new array method

- `pydash.arrays.duplicates()`

- 2 new function methods

- `pydash.functions.ary()`
 - `pydash.functions.rearg()`

- 1 new collection method:

- `pydash.collections.sort_by_all()`

- 4 new object methods

- `pydash.objects.to_boolean()`
 - `pydash.objects.to_dict()`
 - `pydash.objects.to_number()`
 - `pydash.objects.to_plain_object()`

- 4 new predicate methods

- `pydash.predicates.is_blank()`
 - `pydash.predicates.is_builtin()` and alias `pydash.predicates.is_native()`
 - `pydash.predicates.is_match()`
 - `pydash.predicates.is_tuple()`

- 1 new utility method
 - `pydash.utilities.prop_of()` and alias `pydash.utilities.property_of()`
- 6 new aliases:
 - `pydash.predicates.is_bool()` for `pydash.predicates.is_boolean()`
 - `pydash.predicates.is_dict()` for `pydash.predicates.is_plain_object()`
 - `pydash.predicates.is_int()` for `pydash.predicates.is_integer()`
 - `pydash.predicates.is_num()` for `pydash.predicates.is_number()`
 - `pydash.strings.truncate()` for `pydash.strings.trunc()`
 - `pydash.strings.underscore_case()` for `pydash.strings.snake_case()`
- Chaining can now accept the root value argument late.
- Chains can be re-used with different initial values via `chain().plant()`.
- New chains can be created using the chain's computed value as the new chain's initial value via `chain().commit()`.
- Support iteration over class instance properties for non-list, non-dict, and non-iterable objects.

Late Value Chaining

The passing of the root `value` argument for chaining can now be done “late” meaning that you can build chains without providing a value at the beginning. This allows you to build a chain and re-use it with different root values:

```
>>> from pydash import py_
>>> square_sum = py_().power(2).sum()
>>> [square_sum([1, 2, 3]), square_sum([4, 5, 6]), square_sum([7, 8, 9])]
[14, 77, 194]
```

See also:

- For more details on method chaining, check out [Method Chaining](#).
- For a full listing of changes in v3.0.0, check out the [Changelog](#).

From v1.x.x to v2.0.0

There were several breaking and potentially breaking changes in v2.0.0:

- `pydash.arrays.flatten()` is now shallow by default. Previously, it was deep by default. For deep flattening, use either `flatten(..., is_deep=True)` or `flatten_deep(...)`.
- `pydash.predicates.is_number()` now returns `False` for boolean `True` and `False`. Previously, it returned `True`.
- Internally, the files located in `pydash.api` were moved to `pydash`. If you imported from `pydash.api.<module>`, then it's recommended to change your imports to pull from `pydash`.
- The function `functions()` was renamed to `callablees()` to avoid ambiguities with the module `functions.py`.

Some notable new features:

- Callback functions no longer require the full call signature definition.
- A new “_” instance was added which supports both method chaining and module method calling. See [py_Instance](#) for more details.

See also:

For a full listing of changes in v2.0.0, check out the [Changelog](#).

Includes links to source code.

API Reference

All public functions are available from the main module.

```
import pydash  
  
pydash.<function>
```

This is the recommended way to use pydash.

```
# OK (importing main module)  
import pydash  
pydash.where({})  
  
# OK (import from main module)  
from pydash import where  
where({})  
  
# NOT RECOMMENDED (importing from submodule)  
from pydash.collections import where
```

Only the main pydash module API is guaranteed to adhere to semver. It's possible that backwards incompatibility outside the main module API could be broken between minor releases.

py_ Instance

There is a special `py_` instance available from `pydash` that supports method calling and method chaining from a single object:

```
from pydash import py_

# Method calling
py_.initial([1, 2, 3, 4, 5]) == [1, 2, 3, 4]

# Method chaining
py_([1, 2, 3, 4, 5]).initial().value() == [1, 2, 3, 4]

# Method aliasing to underscore suffixed methods that shadow builtin names
py_.map is py_.map_
py_([1, 2, 3]).map(_.to_string).value() == py_([1, 2, 3]).map_(_.to_string).value()
```

The `py_` instance is basically a combination of using `pydash.<function>` and `pydash.chain`.

A full listing of aliased `py_` methods:

- `_.object` is `pydash.arrays.object_()`
- `_.slice` is `pydash.arrays.slice_()`
- `_.zip` is `pydash.arrays.zip_()`
- `_.all` is `pydash.collections.all_()`
- `_.any` is `pydash.collections.any_()`
- `_.filter` is `pydash.collections.filter_()`
- `_.map` is `pydash.collections.map_()`
- `_.max` is `pydash.collections.max_()`
- `_.min` is `pydash.collections.min_()`
- `_.reduce` is `pydash.collections.reduce_()`
- `_.pow` is `pydash.numerical.pow_()`
- `_.round` is `pydash.numerical.round_()`
- `_.sum` is `pydash.numerical.sum_()`
- `_.property` is `pydash.utilities.property_()`
- `_.range` is `pydash.utilities.range_()`

Arrays

Functions that operate on lists.

New in version 1.0.0.

`pydash.arrays.chunk(array, size=1)`

Creates a list of elements split into groups the length of *size*. If *array* can't be split evenly, the final chunk will be the remaining elements.

Parameters

- **array** (*list*) – List to chunk.
- **size** (*int, optional*) – Chunk size. Defaults to 1.

Returns New list containing chunks of *array*.

Return type list

Example

```
>>> chunk([1, 2, 3, 4, 5], 2)
[[1, 2], [3, 4], [5]]
```

New in version 1.1.0.

`pydash.arrays.compact(array)`

Creates a list with all falsey values of array removed.

Parameters `array` (*list*) – List to compact.

Returns Compacted list.

Return type list

Example

```
>>> compact(['', 1, 0, True, False, None])
[1, True]
```

New in version 1.0.0.

`pydash.arrays.concat(*arrays)`

Concatenates zero or more lists into one.

Parameters `arrays` (*list*) – Lists to concatenate.

Returns Concatenated list.

Return type list

Example

```
>>> concat([1, 2], [3, 4], [[5], [6]])
[1, 2, 3, 4, 5, 6]
```

New in version 2.0.0.

Changed in version 4.0.0: Renamed from `cat` to `concat`.

`pydash.arrays.difference(array, *others)`

Creates a list of list elements not present in others.

Parameters

- **array** (*list*) – List to process.
- **others** (*list*) – Lists to check.

Returns Difference between *others*.

Return type list

Example

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

New in version 1.0.0.

`pydash.arrays.difference_by(array, *others, **kargs)`

This method is like `difference()` except that it accepts an *iteratee* which is invoked for each element of each array to generate the criterion by which they're compared. The order and references of result values are determined by *array*. The *iteratee* is invoked with one argument: (*value*).

Parameters

- **array** (*list*) – The array to find the difference of.
- **others** (*list*) – Lists to check for difference with *array*.

Keyword Arguments **iteratee** (*mixed, optional*) – Function to transform the elements of the arrays. Defaults to `identity()`.

Returns Difference between *others*.

Return type `list`

Example

```
>>> difference_by([1.2, 1.5, 1.7, 2.8], [0.9, 3.2], round)
[1.5, 1.7]
```

New in version 4.0.0.

`pydash.arrays.difference_with(array, *others, **kargs)`

This method is like `difference()` except that it accepts a *comparator* which is invoked to compare the elements of all arrays. The order and references of result values are determined by the first array. The *comparator* is invoked with two arguments: (*arr_val*, *oth_val*).

Parameters

- **array** (*list*) – The array to find the difference of.
- **others** (*list*) – Lists to check for difference with *array*.

Keyword Arguments **comparator** (*callable, optional*) – Function to compare the elements of the arrays. Defaults to `is_equal()`.

Returns Difference between *others*.

Return type `list`

Example

```
>>> array = ['apple', 'banana', 'pear']
>>> others = (['avocado', 'pumpkin'], ['peach'])
>>> comparator = lambda a, b: a[0] == b[0]
>>> difference_with(array, *others, comparator=comparator)
['banana']
```

New in version 4.0.0.

`pydash.arrays.drop(array, n=1)`

Creates a slice of *array* with *n* elements dropped from the beginning.

Parameters

- **array** (*list*) – List to process.
- **n** (*int, optional*) – Number of elements to drop. Defaults to 1.

Returns Dropped list.

Return type list

Example

```
>>> drop([1, 2, 3, 4], 2)
[3, 4]
```

New in version 1.0.0.

Changed in version 1.1.0: Added *n* argument and removed as alias of `rest()`.

Changed in version 3.0.0: Made *n* default to 1.

`pydash.arrays.drop_right(array, n=1)`

Creates a slice of *array* with *n* elements dropped from the end.

Parameters

- **array** (*list*) – List to process.
- **n** (*int, optional*) – Number of elements to drop. Defaults to 1.

Returns Dropped list.

Return type list

Example

```
>>> drop_right([1, 2, 3, 4], 2)
[1, 2]
```

New in version 1.1.0.

Changed in version 3.0.0: Made *n* default to 1.

`pydash.arrays.drop_right_while(array, predicate=None)`

Creates a slice of *array* excluding elements dropped from the end. Elements are dropped until the *predicate* returns falsey. The *predicate* is invoked with three arguments: (*value*, *index*, *array*).

Parameters

- **array** (*list*) – List to process.
- **predicate** (*mixed*) – Predicate called per iteration

Returns Dropped list.

Return type list

Example

```
>>> drop_right_while([1, 2, 3, 4], lambda x: x >= 3)
[1, 2]
```

New in version 1.1.0.

`pydash.arrays.drop_while(array, predicate=None)`

Creates a slice of *array* excluding elements dropped from the beginning. Elements are dropped until the *predicate* returns falsey. The *predicate* is invoked with three arguments: (value, index, array).

Parameters

- **array** (*list*) – List to process.
- **predicate** (*mixed*) – Predicate called per iteration

Returns Dropped list.

Return type list

Example

```
>>> drop_while([1, 2, 3, 4], lambda x: x < 3)
[3, 4]
```

New in version 1.1.0.

`pydash.arrays.duplicates(array, iteratee=None)`

Creates a unique list of duplicate values from *array*. If *iteratee* is passed, each element of *array* is passed through a *iteratee* before duplicates are computed. The *iteratee* is invoked with three arguments: (value, index, array). If an object path is passed for *iteratee*, the created *iteratee* will return the path value of the given element. If an object is passed for *iteratee*, the created filter style *iteratee* will return `True` for elements that have the properties of the given object, else `False`.

Parameters

- **array** (*list*) – List to process.
- **iteratee** (*mixed, optional*) – Iteratee applied per iteration.

Returns List of duplicates.

Return type list

Example

```
>>> duplicates([0, 1, 3, 2, 3, 1])
[3, 1]
```

New in version 3.0.0.

`pydash.arrays.fill(array, value, start=0, end=None)`

Fills elements of *array* with *value* from *start* up to, but not including, *end*.

Parameters

- **array** (*list*) – List to fill.

- **value** (*mixed*) – Value to fill with.
- **start** (*int*, *optional*) – Index to start filling. Defaults to 0.
- **end** (*int*, *optional*) – Index to end filling. Defaults to `len(array)`.

Returns Filled *array*.

Return type list

Example

```
>>> fill([1, 2, 3, 4, 5], 0)
[0, 0, 0, 0, 0]
>>> fill([1, 2, 3, 4, 5], 0, 1, 3)
[1, 0, 0, 4, 5]
>>> fill([1, 2, 3, 4, 5], 0, 0, 100)
[0, 0, 0, 0, 0]
```

Warning: *array* is modified in place.

New in version 3.1.0.

`pydash.arrays.find_index(array, predicate=None)`

This method is similar to `pydash.collections.find()`, except that it returns the index of the element that passes the predicate check, instead of the element itself.

Parameters

- **array** (*list*) – List to process.
- **predicate** (*mixed*, *optional*) – Predicate applied per iteration.

Returns Index of found item or `-1` if not found.

Return type int

Example

```
>>> find_index([1, 2, 3, 4], lambda x: x >= 3)
2
>>> find_index([1, 2, 3, 4], lambda x: x > 4)
-1
```

New in version 1.0.0.

`pydash.arrays.find_last_index(array, predicate=None)`

This method is similar to `find_index()`, except that it iterates over elements from right to left.

Parameters

- **array** (*list*) – List to process.
- **predicate** (*mixed*, *optional*) – Predicate applied per iteration.

Returns Index of found item or `-1` if not found.

Return type int

Example

```
>>> find_last_index([1, 2, 3, 4], lambda x: x >= 3)
3
>>> find_index([1, 2, 3, 4], lambda x: x > 4)
-1
```

New in version 1.0.0.

`pydash.arrays.flatten(array)`

Flattens a nested array. If *is_deep* is `True` the array is recursively flattened, otherwise it is only flattened a single level.

Parameters `array (list)` – List to flatten.

Returns Flattened list.

Return type list

Example

```
>>> flatten([[1], [2, [3]], [[4]]])
[1, 2, [3], [4]]
```

New in version 1.0.0.

Changed in version 2.0.0: Removed *callback* option. Added *is_deep* option. Made it shallow by default.

Changed in version 4.0.0: Removed *is_deep* option. Use *flatten_deep()* instead.

`pydash.arrays.flatten_deep(array)`

Flattens a nested array recursively. This is the same as calling `flatten(array, is_deep=True)`.

Parameters `array (list)` – List to flatten.

Returns Flattened list.

Return type list

Example

```
>>> flatten_deep([[1], [2, [3]], [[4]]])
[1, 2, 3, 4]
```

New in version 2.0.0.

`pydash.arrays.flatten_depth(array, depth=1)`

Recursively flatten *array* up to *depth* times.

Parameters

- **array (list)** – List to flatten.
- **depth (int, optional)** – Depth to flatten to. Defaults to 1.

Returns Flattened list.

Return type list

Example

```
>>> flatten_depth([[1], [2, [3]], [[4]]], 1)
[[1], [2, [3]], [[4]]]
>>> flatten_depth([[1], [2, [3]], [[4]]], 2)
[1, 2, [3], [4]]
>>> flatten_depth([[1], [2, [3]], [[4]]], 3)
[1, 2, 3, 4]
>>> flatten_depth([[1], [2, [3]], [[4]]], 4)
[1, 2, 3, 4]
```

New in version 4.0.0.

`pydash.arrays.from_pairs(pairs)`

Returns a dict from the given list of pairs.

Parameters `pairs` (*list*) – List of key-value pairs.

Returns dict

Example

```
>>> from_pairs(['a', 1])
{'a': 1}
```

New in version 4.0.0.

`pydash.arrays.head(array)`

Return the first element of *array*.

Parameters `array` (*list*) – List to process.

Returns First element of list.

Return type mixed

Example

```
>>> head([1, 2, 3, 4])
1
```

New in version 1.0.0.

Changed in version Renamed: from `first` to `head`.

`pydash.arrays.index_of(array, value, from_index=0)`

Gets the index at which the first occurrence of value is found.

Parameters

- **array** (*list*) – List to search.
- **value** (*mixed*) – Value to search for.
- **from_index** (*int, optional*) – Index to search from.

Returns Index of found item or -1 if not found.

Return type int

Example

```
>>> index_of([1, 2, 3, 4], 2)
1
>>> index_of([2, 1, 2, 3], 2, from_index=1)
2
```

New in version 1.0.0.

`pydash.arrays.initial(array)`

Return all but the last element of *array*.

Parameters *array* (*list*) – List to process.

Returns Initial part of *array*.

Return type *list*

Example

```
>>> initial([1, 2, 3, 4])
[1, 2, 3]
```

New in version 1.0.0.

`pydash.arrays.intercalate(array, separator)`

Like *intersperse()* for lists of lists but shallowly flattening the result.

Parameters

- **array** (*list*) – List to intercalate.
- **separator** (*mixed*) – Element to insert.

Returns Intercalated list.

Return type *list*

Example

```
>>> intercalate([1, [2], [3], 4], 'x')
[1, 'x', 2, 'x', 3, 'x', 4]
```

New in version 2.0.0.

`pydash.arrays.interleave(*arrays)`

Merge multiple lists into a single list by inserting the next element of each list by sequential round-robin into the new list.

Parameters *arrays* (*list*) – Lists to interleave.

Retruns: *list*: Interleaved list.

Example

```
>>> interleave([1, 2, 3], [4, 5, 6], [7, 8, 9])
[1, 4, 7, 2, 5, 8, 3, 6, 9]
```

New in version 2.0.0.

`pydash.arrays.intersection(array, *others)`

Computes the intersection of all the passed-in arrays.

Parameters

- **array** (*list*) – The array to find the intersection of.
- **others** (*list*) – Lists to check for intersection with *array*.

Returns Intersection of provided lists.

Return type list

Example

```
>>> intersection([1, 2, 3], [1, 2, 3, 4, 5], [2, 3])
[2, 3]
```

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

New in version 1.0.0.

Changed in version 4.0.0: Support finding intersection of unhashable types.

`pydash.arrays.intersection_by(array, *others, **kwargs)`

This method is like `intersection()` except that it accepts an iteratee which is invoked for each element of each array to generate the criterion by which they're compared. The order and references of result values are determined by *array*. The iteratee is invoked with one argument: (*value*).

Parameters

- **array** (*list*) – The array to find the intersection of.
- **others** (*list*) – Lists to check for intersection with *array*.

Keyword Arguments **iteratee** (*mixed, optional*) – Function to transform the elements of the arrays. Defaults to `identity()`.

Returns Intersection of provided lists.

Return type list

Example

```
>>> intersection_by([1.2, 1.5, 1.7, 2.8], [0.9, 3.2], round)
[1.2, 2.8]
```

New in version 4.0.0.

`pydash.arrays.intersection_with(array, *others, **kwargs)`

This method is like `intersection()` except that it accepts a comparator which is invoked to compare the elements of all arrays. The order and references of result values are determined by the first array. The comparator is invoked with two arguments: (`arr_val`, `oth_val`).

Parameters

- **array** (*list*) – The array to find the intersection of.
- **others** (*list*) – Lists to check for intersection with *array*.

Keyword Arguments **comparator** (*callable*, *optional*) – Function to compare the elements of the arrays. Defaults to `is_equal()`.

Returns Intersection of provided lists.

Return type `list`

Example

```
>>> array = ['apple', 'banana', 'pear']
>>> others = (['avocado', 'pumpkin'], ['peach'])
>>> comparator = lambda a, b: a[0] == b[0]
>>> intersection_with(array, *others, comparator=comparator)
['pear']
```

New in version 4.0.0.

`pydash.arrays.intersperse(array, separator)`

Insert a separating element between the elements of *array*.

Parameters

- **array** (*list*) – List to intersperse.
- **separator** (*mixed*) – Element to insert.

Returns Interspersed list.

Return type `list`

Example

```
>>> intersperse([1, [2], [3], 4], 'x')
[1, 'x', [2], 'x', [3], 'x', 4]
```

New in version 2.0.0.

`pydash.arrays.last(array)`

Return the last element of *array*.

Parameters **array** (*list*) – List to process.

Returns Last part of *array*.

Return type `mixed`

Example

```
>>> last([1, 2, 3, 4])
4
```

New in version 1.0.0.

`pydash.arrays.last_index_of(array, value, from_index=None)`

Gets the index at which the last occurrence of value is found.

Parameters

- **array** (*list*) – List to search.
- **value** (*mixed*) – Value to search for.
- **from_index** (*int, optional*) – Index to search from.

Returns Index of found item or `False` if not found.

Return type `int`

Example

```
>>> last_index_of([1, 2, 2, 4], 2)
2
>>> last_index_of([1, 2, 2, 4], 2, from_index=1)
1
```

New in version 1.0.0.

`pydash.arrays.mapcat(array, iteratee=None)`

Map an iteratee to each element of a list and concatenate the results into a single list using `cat()`.

Parameters

- **array** (*list*) – List to map and concatenate.
- **iteratee** (*mixed*) – Iteratee to apply to each element.

Returns Mapped and concatenated list.

Return type `list`

Example

```
>>> mapcat(range(4), lambda x: list(range(x)))
[0, 0, 1, 0, 1, 2]
```

New in version 2.0.0.

`pydash.arrays.nth(array, pos=0)`

Gets the element at index `n` of array.

Parameters

- **array** (*list*) – List passed in by the user.
- **pos** (*int*) – Index of element to return.

Returns Returns the element at `pos`.

Return type `mixed`

Example

```
>>> nth([1, 2, 3], 0)
1
>>> nth([3, 4, 5, 6], 2)
5
>>> nth([11, 22, 33], -1)
33
>>> nth([11, 22, 33])
11
```

New in version 4.0.0.

`pydash.arrays.pull(array, *values)`

Removes all provided values from the given array.

Parameters

- **array** (*list*) – List to pull from.
- **values** (*mixed*) – Values to remove.

Returns Modified *array*.

Return type `list`

Warning: *array* is modified in place.

Example

```
>>> pull([1, 2, 2, 3, 3, 4], 2, 3)
[1, 4]
```

New in version 1.0.0.

Changed in version 4.0.0: `pull()` method now calls `pull_all()` method for the desired functionality.

`pydash.arrays.pull_all(array, values)`

Removes all provided values from the given array.

Parameters

- **array** (*list*) – Array to modify.
- **values** (*list*) – Values to remove.

Returns Modified *array*.

Return type `list`

Example

```
>>> pull_all([1, 2, 2, 3, 3, 4], [2, 3])
[1, 4]
```

New in version 4.0.0.

`pydash.arrays.pull_all_by(array, values, iteratee=None)`

This method is like `pull_all()` except that it accepts `iteratee` which is invoked for each element of `array` and `values` to generate the criterion by which they're compared. The `iteratee` is invoked with one argument: `(value)`.

Parameters

- **array** (*list*) – Array to modify.
- **values** (*list*) – Values to remove.
- **iteratee** (*mixed, optional*) – Function to transform the elements of the arrays. Defaults to `identity()`.

Returns Modified *array*.

Return type *list*

Example

```
>>> array = [{'x': 1}, {'x': 2}, {'x': 3}, {'x': 1}]
>>> pull_all_by(array, [{'x': 1}, {'x': 3}], 'x')
[{'x': 2}]
```

New in version 4.0.0.

`pydash.arrays.pull_all_with(array, values, comparator=None)`

This method is like `pull_all()` except that it accepts `comparator` which is invoked to compare elements of `array` to `values`. The `comparator` is invoked with two arguments: `(arr_val, oth_val)`.

Parameters

- **array** (*list*) – Array to modify.
- **values** (*list*) – Values to remove.
- **comparator** (*callable, optional*) – Function to compare the elements of the arrays. Defaults to `is_equal()`.

Returns Modified *array*.

Return type *list*

Example

```
>>> array = [{'x': 1, 'y': 2}, {'x': 3, 'y': 4}, {'x': 5, 'y': 6}]
>>> res = pull_all_with(array, [{'x': 3, 'y': 4}], lambda a, b: a == b)
>>> res == [{'x': 1, 'y': 2}, {'x': 5, 'y': 6}]
True
>>> array = [{'x': 1, 'y': 2}, {'x': 3, 'y': 4}, {'x': 5, 'y': 6}]
>>> res = pull_all_with(array, [{'x': 3, 'y': 4}], lambda a, b: a != b)
```

```
>>> res == [{'x': 3, 'y': 4}]
True
```

New in version 4.0.0.

`pydash.arrays.pull_at(array, *indexes)`

Removes elements from *array* corresponding to the specified indexes and returns a list of the removed elements. Indexes may be specified as a list of indexes or as individual arguments.

Parameters

- **array** (*list*) – List to pull from.
- **indexes** (*int*) – Indexes to pull.

Returns Modified *array*.

Return type list

Warning: *array* is modified in place.

Example

```
>>> pull_at([1, 2, 3, 4], 0, 2)
[2, 4]
```

New in version 1.1.0.

`pydash.arrays.push(array, *items)`

Push items onto the end of *array* and return modified *array*.

Parameters

- **array** (*list*) – List to push to.
- **items** (*mixed*) – Items to append.

Returns Modified *array*.

Return type list

Warning: *array* is modified in place.

Example

```
>>> array = [1, 2, 3]
>>> push(array, 4, 5, [6])
[1, 2, 3, 4, 5, [6]]
```

New in version 2.2.0.

Changed in version 4.0.0: Removed alias `append`.

`pydash.arrays.remove(array, predicate=None)`

Removes all elements from a list that the predicate returns truthy for and returns an array of removed elements.

Parameters

- **array** (*list*) – List to remove elements from.
- **predicate** (*mixed, optional*) – Predicate applied per iteration.

Returns Removed elements of *array*.

Return type list

Warning: *array* is modified in place.

Example

```
>>> array = [1, 2, 3, 4]
>>> items = remove(array, lambda x: x >= 3)
>>> items
[3, 4]
>>> array
[1, 2]
```

New in version 1.0.0.

`pydash.arrays.reverse(array)`

Return *array* in reverse order.

Parameters **array** (*list/string*) – Object to process.

Returns Reverse of object.

Return type list/string

Example

```
>>> reverse([1, 2, 3, 4])
[4, 3, 2, 1]
```

New in version 2.2.0.

`pydash.arrays.shift(array)`

Remove the first element of *array* and return it.

Parameters **array** (*list*) – List to shift.

Returns First element of *array*.

Return type mixed

Warning: *array* is modified in place.

Example

```
>>> array = [1, 2, 3, 4]
>>> item = shift(array)
>>> item
1
>>> array
[2, 3, 4]
```

New in version 2.2.0.

`pydash.arrays.slice_(array, start=0, end=None)`

Slices *array* from the *start* index up to, but not including, the *end* index.

Parameters

- **array** (*list*) – Array to slice.
- **start** (*int, optional*) – Start index. Defaults to 0.
- **end** (*int, optional*) – End index. Defaults to selecting the value at *start* index.

Returns Sliced list.

Return type `list`

Example

```
>>> slice_([1, 2, 3, 4])
[1]
>>> slice_([1, 2, 3, 4], 1)
[2]
>>> slice_([1, 2, 3, 4], 1, 3)
[2, 3]
```

New in version 1.1.0.

`pydash.arrays.sort_(array, comparator=None, key=None, reverse=False)`

Sort *array* using optional *comparator*, *key*, and *reverse* options and return sorted *array*.

Note: Python 3 removed the option to pass a custom comparator function and instead only allows a key function. Therefore, if a comparator function is passed in, it will be converted to a key function automatically using `functools.cmp_to_key`.

Parameters

- **array** (*list*) – List to sort.
- **comparator** (*callable, optional*) – A custom comparator function used to sort the list. Function should accept two arguments and return a negative, zero, or position number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument. Defaults to `None`. This argument is mutually exclusive with *key*.
- **key** (*iteratee, optional*) – A function of one argument used to extract a a comparator key from each list element. Defaults to `None`. This argument is mutually exclusive with *comparator*.
- **reverse** (*bool, optional*) – Whether to reverse the sort. Defaults to `False`.

Returns Sorted list.

Return type list

Warning: *array* is modified in place.

Example

```
>>> sort([2, 1, 4, 3])
[1, 2, 3, 4]
>>> sort([2, 1, 4, 3], reverse=True)
[4, 3, 2, 1]
>>> results = sort([{'a': 2, 'b': 1}, {'a': 3, 'b': 2},
↳                               {'a': 0, 'b': 3}],
↳key=lambda item: item['a'])
>>> assert results == [{'a': 0, 'b': 3}, {'a': 2, 'b':
↳ 1}, {'a': 3, 'b': 2}]
```

New in version 2.2.0.

`pydash.arrays.sorted_index(array, value)`

Uses a binary search to determine the lowest index at which *value* should be inserted into *array* in order to maintain its sort order.

Parameters

- **array** (*list*) – List to inspect.
- **value** (*mixed*) – Value to evaluate.

Returns

Returns the index at which *value* should be inserted into *array*.

Return type int

Example

```
>>> sorted_index([1, 2, 2, 3, 4], 2)
1
```

New in version 1.0.0.

Changed in version 4.0.0: Move iteratee support to `sorted_index_by()`.

`pydash.arrays.sorted_index_by(array, value, iteratee=None)`

This method is like `sorted_index()` except that it accepts *iteratee* which is invoked for *value* and each element of *array* to compute their sort ranking. The *iteratee* is invoked with one argument: (*value*).

Parameters

- **array** (*list*) – List to inspect.
- **value** (*mixed*) – Value to evaluate.
- **iteratee** (*mixed, optional*) – The *iteratee* invoked per element. Defaults to `identity()`.

Returns

Returns the index at which *value* should be inserted into *array*.

Return type int

Example

```
>>> array = [{'x': 4}, {'x': 5}]
>>> sorted_index_by(array, {'x': 4}, lambda o: o['x'])
0
>>> sorted_index_by(array, {'x': 4}, 'x')
0
```

New in version 4.0.0.

`pydash.arrays.sorted_index_of(array, value)`

Returns the index of the matched *value* from the sorted *array*, else `-1`.

Parameters

- **array** (*list*) – Array to inspect.
- **value** (*mixed*) – Value to search for.

Returns Returns the index of the first matched value, else `-1`.

Return type int

Example

```
>>> sorted_index_of([3, 5, 7, 10], 3)
0
>>> sorted_index_of([10, 10, 5, 7, 3], 10)
-1
```

New in version 4.0.0.

`pydash.arrays.sorted_last_index(array, value)`

This method is like `sorted_index()` except that it returns the highest index at which *value* should be inserted into *array* in order to maintain its sort order.

Parameters

- **array** (*list*) – List to inspect.
- **value** (*mixed*) – Value to evaluate.

Returns

Returns the index at which *value* should be inserted into *array*.

Return type int

Example

```
>>> sorted_last_index([1, 2, 2, 3, 4], 2)
3
```

New in version 1.1.0.

Changed in version 4.0.0: Move iteratee support to `sorted_last_index_by()`.

`pydash.arrays.sorted_last_index_by(array, value, iteratee=None)`

This method is like `sorted_last_index()` except that it accepts `iteratee` which is invoked for `value` and each element of `array` to compute their sort ranking. The `iteratee` is invoked with one argument: (`value`).

Parameters

- **array** (*list*) – List to inspect.
- **value** (*mixed*) – Value to evaluate.
- **iteratee** (*mixed, optional*) – The `iteratee` invoked per element. Defaults to `identity()`.

Returns

Returns the index at which `value` should be inserted into `array`.

Return type `int`

Example

```
>>> array = [{'x': 4}, {'x': 5}]
>>> sorted_last_index_by(array, {'x': 4}, lambda o: o['x'])
1
>>> sorted_last_index_by(array, {'x': 4}, 'x')
1
```

`pydash.arrays.sorted_last_index_of(array, value)`

This method is like `last_index_of()` except that it performs a binary search on a sorted `array`.

Parameters

- **array** (*list*) – Array to inspect.
- **value** (*mixed*) – Value to search for.

Returns Returns the index of the matched value, else `-1`.

Return type `int`

Example

```
>>> sorted_last_index_of([4, 5, 5, 5, 6], 5)
3
>>> sorted_last_index_of([6, 5, 5, 5, 4], 6)
-1
```

New in version 4.0.0.

`pydash.arrays.sorted_uniq(array)`

Return sorted array with unique elements.

Parameters **array** (*list*) – List of values to be sorted.

Returns List of unique elements in a sorted fashion.

Return type `list`

Example

```
>>> sorted_uniq([4, 2, 2, 5])
[2, 4, 5]
>>> sorted_uniq([-2, -2, 4, 1])
[-2, 1, 4]
```

New in version 4.0.0.

`pydash.arrays.sorted_uniq_by(array, iteratee=None)`

This method is like `sorted_uniq()` except that it accepts `iteratee` which is invoked for each element in array to generate the criterion by which uniqueness is computed. The order of result values is determined by the order they occur in the array. The `iteratee` is invoked with one argument: `(value)`.

Parameters

- **array** (*list*) – List of values to be sorted.
- **iteratee** (*mixed, optional*) – Function to transform the elements of the arrays. Defaults to `identity()`.

Returns Unique list.

Return type list

Example

```
>>> sorted_uniq_by([3, 2, 1, 3, 2, 1], lambda val: val % 2)
[2, 3]
```

New in version 4.0.0.

`pydash.arrays.splice(array, start, count=None, *items)`

Modify the contents of `array` by inserting elements starting at index `start` and removing `count` number of elements after.

Parameters

- **array** (*list/str*) – List to splice.
- **start** (*int*) – Start to splice at.
- **count** (*int, optional*) – Number of items to remove starting at `start`. If `None` then all items after `start` are removed. Defaults to `None`.
- **items** (*mixed*) – Elements to insert starting at `start`. Each item is inserted in the order given.

Returns The removed elements of `array` or the spliced string.

Return type list/str

Warning: `array` is modified in place if `list`.

Example

```

>>> array = [1, 2, 3, 4]
>>> splice(array, 1)
[2, 3, 4]
>>> array
[1]
>>> array = [1, 2, 3, 4]
>>> splice(array, 1, 2)
[2, 3]
>>> array
[1, 4]
>>> array = [1, 2, 3, 4]
>>> splice(array, 1, 2, 0, 0)
[2, 3]
>>> array
[1, 0, 0, 4]

```

New in version 2.2.0.

Changed in version 3.0.0: Support string splicing.

`pydash.arrays.split_at(array, index)`

Returns a list of two lists composed of the split of *array* at *index*.

Parameters

- **array** (*list*) – List to split.
- **index** (*int*) – Index to split at.

Returns Split list.

Return type list

Example

```

>>> split_at([1, 2, 3, 4], 2)
[[1, 2], [3, 4]]

```

New in version 2.0.0.

`pydash.arrays.tail(array)`

Return all but the first element of *array*.

Parameters **array** (*list*) – List to process.

Returns Rest of the list.

Return type list

Example

```

>>> tail([1, 2, 3, 4])
[2, 3, 4]

```

New in version 1.0.0.

Changed in version 4.0.0: Renamed from `rest` to `tail`.

`pydash.arrays.take(array, n=1)`

Creates a slice of *array* with *n* elements taken from the beginning.

Parameters

- **array** (*list*) – List to process.
- **n** (*int, optional*) – Number of elements to take. Defaults to 1.

Returns Taken list.

Return type list

Example

```
>>> take([1, 2, 3, 4], 2)
[1, 2]
```

New in version 1.0.0.

Changed in version 1.1.0: Added *n* argument and removed as alias of `first()`.

Changed in version 3.0.0: Made *n* default to 1.

`pydash.arrays.take_right(array, n=1)`

Creates a slice of *array* with *n* elements taken from the end.

Parameters

- **array** (*list*) – List to process.
- **n** (*int, optional*) – Number of elements to take. Defaults to 1.

Returns Taken list.

Return type list

Example

```
>>> take_right([1, 2, 3, 4], 2)
[3, 4]
```

New in version 1.1.0.

Changed in version 3.0.0: Made *n* default to 1.

`pydash.arrays.take_right_while(array, predicate=None)`

Creates a slice of *array* with elements taken from the end. Elements are taken until the *predicate* returns falsey. The *predicate* is invoked with three arguments: (*value*, *index*, *array*).

Parameters

- **array** (*list*) – List to process.
- **predicate** (*mixed*) – Predicate called per iteration

Returns Dropped list.

Return type list

Example

```
>>> take_right_while([1, 2, 3, 4], lambda x: x >= 3)
[3, 4]
```

New in version 1.1.0.

`pydash.arrays.take_while(array, predicate=None)`

Creates a slice of *array* with elements taken from the beginning. Elements are taken until the *predicate* returns falsey. The *predicate* is invoked with three arguments: (value, index, array).

Parameters

- **array** (*list*) – List to process.
- **predicate** (*mixed*) – Predicate called per iteration

Returns Taken list.

Return type list

Example

```
>>> take_while([1, 2, 3, 4], lambda x: x < 3)
[1, 2]
```

New in version 1.1.0.

`pydash.arrays.union(array, *others)`

Computes the union of the passed-in arrays.

Parameters

- **array** (*list*) – List to union with.
- **others** (*list*) – Lists to unionize with *array*.

Returns Unionized list.

Return type list

Example

```
>>> union([1, 2, 3], [2, 3, 4], [3, 4, 5])
[1, 2, 3, 4, 5]
```

New in version 1.0.0.

`pydash.arrays.union_by(array, *others, **kargs)`

This method is similar to `union()` except that it accepts *iteratee* which is invoked for each element of each arrays to generate the criterion by which uniqueness is computed.

Parameters

- **array** (*list*) – List to unionize with.
- **others** (*list*) – Lists to unionize with *array*.

Keyword Arguments **iteratee** (*function*) – Function to invoke on each element.

Returns Unionized list.

Return type list

Example

```
>>> union_by([1, 2, 3], [2, 3, 4], iteratee=lambda x: x % 2)
[1, 2]
>>> union_by([1, 2, 3], [2, 3, 4], iteratee=lambda x: x % 9)
[1, 2, 3, 4]
```

New in version 4.0.0.

`pydash.arrays.union_with(array, *others, **kargs)`

This method is like `union()` except that it accepts comparator which is invoked to compare elements of arrays. Result values are chosen from the first array in which the value occurs.

Parameters

- **array** (*list*) – List to unionize with.
- **others** (*list*) – Lists to unionize with *array*.

Keyword Arguments **comparator** (*callable, optional*) – Function to compare the elements of the arrays. Defaults to `is_equal()`.

Returns Unionized list.

Return type list

Example

```
>>> comparator = lambda a, b: (a % 2) == (b % 2)
>>> union_with([1, 2, 3], [2, 3, 4], comparator=comparator)
[1, 2]
>>> union_with([1, 2, 3], [2, 3, 4])
[1, 2, 3, 4]
```

New in version 4.0.0.

`pydash.arrays.uniq(array)`

Creates a duplicate-value-free version of the array. If *iteratee* is passed, each element of array is passed through a *iteratee* before uniqueness is computed. The *iteratee* is invoked with three arguments: (*value*, *index*, *array*). If an object path is passed for *iteratee*, the created *iteratee* will return the path value of the given element. If an object is passed for *iteratee*, the created filter style *iteratee* will return `True` for elements that have the properties of the given object, else `False`.

Parameters **array** (*list*) – List to process.

Returns Unique list.

Return type list

Example

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

New in version 1.0.0.

Changed in version 4.0.0: Moved *iteratee* argument to `uniq_by()`. Removed alias `unique`.

`pydash.arrays.uniq_by(array, iteratee=None)`

This method is like `uniq()` except that it accepts *iteratee* which is invoked for each element in array to generate the criterion by which uniqueness is computed. The order of result values is determined by the order they occur in the array. The *iteratee* is invoked with one argument: `(value)`.

Parameters

- **array** (*list*) – List to process.
- **iteratee** (*mixed, optional*) – Function to transform the elements of the arrays. Defaults to `identity()`.

Returns Unique list.

Return type list

Example

```
>>> uniq_by([1, 2, 3, 1, 2, 3], lambda val: val % 2)
[1, 2]
```

New in version 4.0.0.

`pydash.arrays.uniq_with(array, comparator=None)`

This method is like `_uniq` except that it accepts *comparator* which is invoked to compare elements of array. The order of result values is determined by the order they occur in the array. The *comparator* is invoked with two arguments: `(value, other)`.

Parameters

- **array** (*list*) – List to process.
- **comparator** (*callable, optional*) – Function to compare the elements of the arrays. Defaults to `is_equal()`.

Returns Unique list.

Return type list

Example

```
>>> uniq_with([1, 2, 3, 4, 5], lambda a, b: (a % 2) == (b % 2))
[1, 2]
```

New in version 4.0.0.

`pydash.arrays.unshift(array, *items)`

Insert the given elements at the beginning of *array* and return the modified list.

Parameters

- **array** (*list*) – List to modify.
- **items** (*mixed*) – Items to insert.

Returns Modified list.

Return type list

Warning: *array* is modified in place.

Example

```
>>> array = [1, 2, 3, 4]
>>> unshift(array, -1, -2)
[-1, -2, 1, 2, 3, 4]
>>> array
[-1, -2, 1, 2, 3, 4]
```

New in version 2.2.0.

`pydash.arrays.unzip(array)`

The inverse of `zip_()`, this method splits groups of elements into lists composed of elements from each group at their corresponding indexes.

Parameters **array** (*list*) – List to process.

Returns Unzipped list.

Return type list

Example

```
>>> unzip([[1, 4, 7], [2, 5, 8], [3, 6, 9]])
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

New in version 1.0.0.

`pydash.arrays.unzip_with(array, iteratee=None)`

This method is like `unzip()` except that it accepts a *iteratee* to specify how regrouped values should be combined. The *iteratee* is invoked with four arguments: (*accumulator*, *value*, *index*, *group*).

Parameters

- **array** (*list*) – List to process.
- **iteratee** (*callable*, *optional*) – Function to combine regrouped values.

Returns Unzipped list.

Return type list

Example

```
>>> from pydash import add
>>> unzip_with([[1, 10, 100], [2, 20, 200]], add)
[3, 30, 300]
```

New in version 3.3.0.

`pydash.arrays.without(array, *values)`

Creates an array with all occurrences of the passed values removed.

Parameters

- **array** (*list*) – List to filter.
- **values** (*mixed*) – Values to remove.

Returns Filtered list.

Return type list

Example

```
>>> without([1, 2, 3, 2, 4, 4], 2, 4)
[1, 3]
```

New in version 1.0.0.

`pydash.arrays.xor(array, *lists)`

Creates a list that is the symmetric difference of the provided lists.

Parameters

- **array** (*list*) – List to process.
- ***lists** (*list*) – Lists to xor with.

Returns XOR'd list.

Return type list

Example

```
>>> xor([1, 3, 4], [1, 2, 4], [2])
[3]
```

New in version 1.0.0.

`pydash.arrays.xor_by(array, *lists, **kwargs)`

This method is like `xor()` except that it accepts `iteratee` which is invoked for each element of each arrays to generate the criterion by which they're compared. The order of result values is determined by the order they occur in the arrays. The `iteratee` is invoked with one argument: (`value`).

Parameters

- **array** (*list*) – List to process.
- ***lists** (*list*) – Lists to xor with.

Keyword Arguments **iteratee** (*mixed, optional*) – Function to transform the elements of the arrays. Defaults to `identity()`.

Returns XOR'd list.

Return type list

Example

```
>>> xor_by([2.1, 1.2], [2.3, 3.4], round)
[1.2, 3.4]
>>> xor_by([{'x': 1}], [{'x': 2}, {'x': 1}], 'x')
[{'x': 2}]
```

New in version 4.0.0.

`pydash.arrays.xor_with(array, *lists, **kwargs)`

This method is like `xor()` except that it accepts comparator which is invoked to compare elements of arrays. The order of result values is determined by the order they occur in the arrays. The comparator is invoked with two arguments: (`arr_val`, `oth_val`).

Parameters

- **array** (*list*) – List to process.
- ***lists** (*list*) – Lists to xor with.

Keyword Arguments **comparator** (*callable*, *optional*) – Function to compare the elements of the arrays. Defaults to `is_equal()`.

Returns XOR'd list.

Return type list

Example

```
>>> objects = [{'x': 1, 'y': 2}, {'x': 2, 'y': 1}]
>>> others = [{'x': 1, 'y': 1}, {'x': 1, 'y': 2}]
>>> expected = [{'y': 1, 'x': 2}, {'y': 1, 'x': 1}]
>>> xor_with(objects, others, lambda a, b: a == b) == expected
True
```

New in version 4.0.0.

`pydash.arrays.zip(*arrays)`

Groups the elements of each array at their corresponding indexes. Useful for separate data sources that are coordinated through matching array indexes.

Parameters **arrays** (*list*) – Lists to process.

Returns Zipped list.

Return type list

Example

```
>>> zip_([1, 2, 3], [4, 5, 6], [7, 8, 9])
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

New in version 1.0.0.

`pydash.arrays.zip_object(keys, values=None)`

Creates a dict composed from lists of keys and values. Pass either a single two dimensional list, i.e. `[[key1, value1], [key2, value2]]`, or two lists, one of keys and one of corresponding values.

Parameters

- **keys** (*list*) – Either a list of keys or a list of `[key, value]` pairs.

- **values** (*list*, *optional*) – List of values to zip.

Returns Zipped dict.

Return type dict

Example

```
>>> zip_object([1, 2, 3], [4, 5, 6])
{1: 4, 2: 5, 3: 6}
```

New in version 1.0.0.

Changed in version 4.0.0: Removed alias `object_`.

`pydash.arrays.zip_object_deep(keys, values=None)`

This method is like `zip_object()` except that it supports property paths.

Parameters

- **keys** (*list*) – Either a list of keys or a list of [key, value] pairs.
- **values** (*list*, *optional*) – List of values to zip.

Returns Zipped dict.

Return type dict

Example

```
>>> expected = {'a': {'b': {'c': 1, 'd': 2}}}
>>> zip_object_deep(['a.b.c', 'a.b.d'], [1, 2]) == expected
True
```

New in version 4.0.0.

`pydash.arrays.zip_with(*arrays, **kwargs)`

This method is like `zip()` except that it accepts an *iteratee* to specify how grouped values should be combined. The *iteratee* is invoked with four arguments: (*accumulator*, *value*, *index*, *group*).

Parameters ***arrays** (*list*) – Lists to process.

Keyword Arguments **iteratee** (*function*) – Function to combine grouped values.

Returns Zipped list of grouped elements.

Return type list

Example

```
>>> from pydash import add
>>> zip_with([1, 2], [10, 20], [100, 200], add)
[111, 222]
>>> zip_with([1, 2], [10, 20], [100, 200], iteratee=add)
[111, 222]
```

New in version 3.3.0.

Chaining

Method chaining interface.

New in version 1.0.0.

`pydash.chaining.chain` (*value*=<`pydash.helpers.NoValue` object>)

Creates a `Chain` object which wraps the given value to enable intuitive method chaining. Chaining is lazy and won't compute a final value until `Chain.value()` is called.

Parameters *value* (*mixed*) – Value to initialize chain operations with.

Returns Instance of `Chain` initialized with *value*.

Return type `Chain`

Example

```
>>> chain([1, 2, 3, 4]).map(lambda x: x * 2).sum().value()
20
>>> chain().map(lambda x: x * 2).sum()([1, 2, 3, 4])
20
```

```
>>> summer = chain([1, 2, 3, 4]).sum()
>>> new_summer = summer.plant([1, 2])
>>> new_summer.value()
3
>>> summer.value()
10
```

```
>>> def echo(item): print(item)
>>> summer = chain([1, 2, 3, 4]).for_each(echo).sum()
>>> committed = summer.commit()
1
2
3
4
>>> committed.value()
10
>>> summer.value()
1
2
3
4
10
```

New in version 1.0.0.

Changed in version 2.0.0: Made chaining lazy.

Changed in version 3.0.0: Added support for late passing of *value*. Added `Chain.plant()` for replacing initial chain value. Added `Chain.commit()` for returning a new `Chain` instance initialized with the results from calling `Chain.value()`.

`pydash.chaining.tap` (*value*, *interceptor*)

Invokes *interceptor* with the *value* as the first argument and then returns *value*. The purpose of this method is to “tap into” a method chain in order to perform operations on intermediate results within the chain.

Parameters

- **value** (*mixed*) – Current value of chain operation.
- **interceptor** (*function*) – Function called on *value*.

Returns *value* after *interceptor* call.

Return type `mixed`

Example

```
>>> data = []
>>> def log(value): data.append(value)
>>> chain([1, 2, 3, 4]).map(lambda x: x * 2).tap(log).value()
[2, 4, 6, 8]
>>> data
[[2, 4, 6, 8]]
```

New in version 1.0.0.

`pydash.chaining.thru` (*value*, *interceptor*)

Returns the result of calling *interceptor* on *value*. The purpose of this method is to pass *value* through a function during a method chain.

Parameters

- **value** (*mixed*) – Current value of chain operation.
- **interceptor** (*function*) – Function called with *value*.

Returns Results of `interceptor(value)`.

Return type `mixed`

Example

```
>>> chain([1, 2, 3, 4]).thru(lambda x: x * 2).value()
[1, 2, 3, 4, 1, 2, 3, 4]
```

New in version 2.0.0.

Collections

Functions that operate on lists and dicts.

New in version 1.0.0.

`pydash.collections.at` (*collection*, **indexes*)

Creates a list of elements from the specified indexes, or keys, of the collection. Indexes may be specified as individual arguments or as arrays of indexes.

Parameters

- **collection** (*list|dict*) – Collection to iterate over.
- **indexes** (*mixed*) – The indexes of *collection* to retrieve, specified as individual indexes or arrays of indexes.

Returns filtered list

Return type list

Example

```
>>> at([1, 2, 3, 4], 0, 2)
[1, 3]
>>> at({'a': 1, 'b': 2, 'c': 3, 'd': 4}, 'a', 'c')
[1, 3]
```

New in version 1.0.0.

`pydash.collections.count_by(collection, iteratee=None)`

Creates an object composed of keys generated from the results of running each element of *collection* through the iteratee.

Parameters

- **collection** (*list*/*dict*) – Collection to iterate over.
- **iteratee** (*mixed*, *optional*) – Iteratee applied per iteration.

Returns Dict containing counts by key.

Return type dict

Example

```
>>> results = count_by([1, 2, 1, 2, 3, 4])
>>> assert results == {1: 2, 2: 2, 3: 1, 4: 1}
>>> results = count_by(['a', 'A', 'B', 'b'], lambda x: x.lower())
>>> assert results == {'a': 2, 'b': 2}
>>> results = count_by({'a': 1, 'b': 1, 'c': 3, 'd': 3})
>>> assert results == {1: 2, 3: 2}
```

New in version 1.0.0.

`pydash.collections.every(collection, predicate=None)`

Checks if the predicate returns a truthy value for all elements of a collection. The predicate is invoked with three arguments: (value, index|key, collection). If a property name is passed for predicate, the created *pluck()* style predicate will return the property value of the given element. If an object is passed for predicate, the created *where()* style predicate will return *True* for elements that have the properties of the given object, else *False*.

Parameters

- **collection** (*list*/*dict*) – Collection to iterate over.
- **predicate** (*mixed*, *optional*) – Predicate applied per iteration.

Returns Whether all elements are truthy.

Return type bool

Example

```

>>> every([1, True, 'hello'])
True
>>> every([1, False, 'hello'])
False
>>> every([{'a': 1}, {'a': True}, {'a': 'hello'}], 'a')
True
>>> every([{'a': 1}, {'a': False}, {'a': 'hello'}], 'a')
False
>>> every([{'a': 1}, {'a': 1}], {'a': 1})
True
>>> every([{'a': 1}, {'a': 2}], {'a': 1})
False

```

New in version 1.0.0.

`pydash.collections.filter_(collection, predicate=None)`

Iterates over elements of a collection, returning a list of all elements the predicate returns truthy for.

Parameters

- **collection** (*list/dict*) – Collection to iterate over.
- **predicate** (*mixed, optional*) – Predicate applied per iteration.

Returns Filtered list.

Return type list

Example

```

>>> results = filter_([{'a': 1}, {'b': 2}, {'a': 1, 'b': 3}], {'a': 1})
>>> assert results == [{'a': 1}, {'a': 1, 'b': 3}]
>>> filter_([1, 2, 3, 4], lambda x: x >= 3)
[3, 4]

```

New in version 1.0.0.

Changed in version 4.0.0: Removed alias `select`.

`pydash.collections.find(collection, predicate=None)`

Iterates over elements of a collection, returning the first element that the predicate returns truthy for.

Parameters

- **collection** (*list/dict*) – Collection to iterate over.
- **predicate** (*mixed, optional*) – Predicate applied per iteration.

Returns First element found or `None`.

Return type mixed

Example

```

>>> find([1, 2, 3, 4], lambda x: x >= 3)
3
>>> find([{'a': 1}, {'b': 2}, {'a': 1, 'b': 2}], {'a': 1})
{'a': 1}

```

New in version 1.0.0.

Changed in version 4.0.0: Removed aliases `detect` and `find_where`.

`pydash.collections.find_last(collection, predicate=None)`

This method is like `find()` except that it iterates over elements of a *collection* from right to left.

Parameters

- **collection** (*list* / *dict*) – Collection to iterate over.
- **predicate** (*mixed*, *optional*) – Predicate applied per iteration.

Returns Last element found or `None`.

Return type `mixed`

Example

```
>>> find_last([1, 2, 3, 4], lambda x: x >= 3)
4
>>> results = find_last([{'a': 1}, {'b': 2}, {'a': 1, 'b': 2}],
                        {'a': 1})
>>> assert results == {'a': 1, 'b': 2}
```

New in version 1.0.0.

`pydash.collections.flat_map(collection, iteratee=None)`

Creates a flattened list of values by running each element in *collection* thru *iteratee* and flattening the mapped results. The *iteratee* is invoked with three arguments: (*value*, *index*|*key*, *collection*).

Parameters

- **collection** (*list* / *dict*) – Collection to iterate over.
- **iteratee** (*mixed*, *optional*) – Iteratee applied per iteration.

Returns Flattened mapped list.

Return type `list`

Example

```
>>> duplicate = lambda n: [[n, n]]
>>> flat_map([1, 2], duplicate)
[[1, 1], [2, 2]]
```

New in version 4.0.0.

`pydash.collections.flat_map_deep(collection, iteratee=None)`

This method is like `flat_map()` except that it recursively flattens the mapped results.

Parameters

- **collection** (*list* / *dict*) – Collection to iterate over.
- **iteratee** (*mixed*, *optional*) – Iteratee applied per iteration.

Returns Flattened mapped list.

Return type `list`

Example

```
>>> duplicate = lambda n: [[n, n]]
>>> flat_map_deep([1, 2], duplicate)
[1, 1, 2, 2]
```

New in version 4.0.0.

`pydash.collections.flat_map_depth(collection, iteratee=None, depth=1)`

This method is like `flat_map()` except that it recursively flattens the mapped results up to *depth* times.

Parameters

- **collection** (*list*/*dict*) – Collection to iterate over.
- **iteratee** (*mixed*, *optional*) – Iteratee applied per iteration.

Returns Flattened mapped list.

Return type list

Example

```
>>> duplicate = lambda n: [[n, n]]
>>> flat_map_depth([1, 2], duplicate, 1)
[[1, 1], [2, 2]]
>>> flat_map_depth([1, 2], duplicate, 2)
[1, 1, 2, 2]
```

New in version 4.0.0.

`pydash.collections.for_each(collection, iteratee=None)`

Iterates over elements of a collection, executing the iteratee for each element.

Parameters

- **collection** (*list*/*dict*) – Collection to iterate over.
- **iteratee** (*mixed*, *optional*) – Iteratee applied per iteration.

Returns *collection*

Return type listdict

Example

```
>>> results = {}
>>> def cb(x): results[x] = x ** 2
>>> for_each([1, 2, 3, 4], cb)
[1, 2, 3, 4]
>>> assert results == {1: 1, 2: 4, 3: 9, 4: 16}
```

New in version 1.0.0.

Changed in version 4.0.0: Removed alias `each`.

`pydash.collections.for_each_right(collection, iteratee)`

This method is like `for_each()` except that it iterates over elements of a *collection* from right to left.

Parameters

- **collection** (*list* / *dict*) – Collection to iterate over.
- **iteratee** (*mixed*, *optional*) – Iteratee applied per iteration.

Returns *collection*

Return type listdict

Example

```
>>> results = {'total': 1}
>>> def cb(x): results['total'] = x * results['total']
>>> for_each_right([1, 2, 3, 4], cb)
[1, 2, 3, 4]
>>> assert results == {'total': 24}
```

New in version 1.0.0.

Changed in version 4.0.0: Removed alias `each_right`.

`pydash.collections.group_by` (*collection*, *iteratee=None*)

Creates an object composed of keys generated from the results of running each element of a *collection* through the iteratee.

Parameters

- **collection** (*list* / *dict*) – Collection to iterate over.
- **iteratee** (*mixed*, *optional*) – Iteratee applied per iteration.

Returns Results of grouping by *iteratee*.

Return type dict

Example

```
>>> results = group_by([{'a': 1, 'b': 2}, {'a': 3, 'b': 4}], 'a')
>>> assert results == {1: [{'a': 1, 'b': 2}], 3: [{'a': 3, 'b': 4}]}
>>> results = group_by([{'a': 1, 'b': 2}, {'a': 3, 'b': 4}], {'a': 1})
>>> assert results == {False: [{'a': 3, 'b': 4}],
↪ True: [{'a': 1, 'b': 2}]}
```

New in version 1.0.0.

`pydash.collections.includes` (*collection*, *target*, *from_index=0*)

Checks if a given value is present in a collection. If *from_index* is negative, it is used as the offset from the end of the collection.

Parameters

- **collection** (*list* / *dict*) – Collection to iterate over.
- **target** (*mixed*) – Target value to compare to.
- **from_index** (*int*, *optional*) – Offset to start search from.

Returns Whether *target* is in *collection*.

Return type bool

Example

```
>>> includes([1, 2, 3, 4], 2)
True
>>> includes([1, 2, 3, 4], 2, from_index=2)
False
>>> includes({'a': 1, 'b': 2, 'c': 3, 'd': 4}, 2)
True
```

New in version 1.0.0.

Changed in version 4.0.0: Renamed from `contains` to `includes` and removed alias `include`.

`pydash.collections.invoke_map(collection, path, *args, **kwargs)`

Invokes the method at *path* of each element in *collection*, returning a list of the results of each invoked method. Any additional arguments are provided to each invoked method. If *path* is a function, it's invoked for each element in *collection*.

Parameters

- **collection** (*list*/*dict*) – Collection to iterate over.
- **path** (*str*/*func*) – String path to method to invoke or callable to invoke for each element in *collection*.
- **args** (*optional*) – Arguments to pass to method call.
- **kwargs** (*optional*) – Keyword arguments to pass to method call.

Returns List of results of invoking method of each item.

Return type list

Example

```
>>> items = [{'a': [{'b': 1}]}, {'a': [{'c': 2}]}]
>>> expected = [{'b': 1}.items(), {'c': 2}.items()]
>>> invoke_map(items, 'a[0].items') == expected
True
```

New in version 4.0.0.

`pydash.collections.key_by(collection, iteratee=None)`

Creates an object composed of keys generated from the results of running each element of the collection through the given *iteratee*.

Parameters

- **collection** (*list*/*dict*) – Collection to iterate over.
- **iteratee** (*mixed*, *optional*) – Iteratee applied per iteration.

Returns Results of indexing by *iteratee*.

Return type dict

Example

```
>>> results = key_by([{'a': 1, 'b': 2}, {'a': 3, 'b': 4}], 'a')
>>> assert results == {1: {'a': 1, 'b': 2}, 3: {'a': 3, 'b': 4}}
```

New in version 1.0.0.

Changed in version 4.0.0: Renamed from `index_by` to `key_by`.

`pydash.collections.map_(collection, iteratee=None)`

Creates an array of values by running each element in the collection through the iteratee. The iteratee is invoked with three arguments: (value, index|key, collection). If a property name is passed for iteratee, the created `pluck()` style iteratee will return the property value of the given element. If an object is passed for iteratee, the created `where()` style iteratee will return `True` for elements that have the properties of the given object, else `False`.

Parameters

- **collection** (*list/dict*) – Collection to iterate over.
- **iteratee** (*mixed, optional*) – Iteratee applied per iteration.

Returns Mapped list.

Return type list

Example

```
>>> map_([1, 2, 3, 4], str)
['1', '2', '3', '4']
>>> map_([{'a': 1, 'b': 2}, {'a': 3, 'b': 4}, {'a': 5, 'b': 6}], 'a')
[1, 3, 5]
>>> map_([[0, 1]], [[2, 3]], [[4, 5]]], '0.1')
[1, 3, 5]
>>> map_([{'a': {'b': 1}}, {'a': {'b': 2}}], 'a.b')
[1, 2]
>>> map_([{'a': {'b': [0, 1]}}, {'a': {'b': [2, 3]}}], 'a.b[1]')
[1, 3]
```

New in version 1.0.0.

Changed in version 4.0.0: Removed alias `collect`.

`pydash.collections.order_by(collection, keys, orders=None, reverse=False)`

This method is like `sort_by()` except that it sorts by key names instead of an iteratee function. Keys can be sorted in descending order by prepending a `"-"` to the key name (e.g. `"name"` would become `"-name"`) or by passing a list of boolean sort options via `orders` where `True` is ascending and `False` is descending.

Parameters

- **collection** (*list/dict*) – Collection to iterate over.
- **keys** (*list*) – List of keys to sort by. By default, keys will be sorted in ascending order. To sort a key in descending order, prepend a `"-"` to the key name. For example, to sort the key value for `"name"` in descending order, use `"-name"`.
- **orders** (*list, optional*) – List of boolean sort orders to apply for each key. `True` corresponds to ascending order while `False` is descending. Defaults to `None`.
- **reverse** (*bool, optional*) – Whether to reverse the sort. Defaults to `False`.

Returns Sorted list.

Return type list

Example

```
>>> items = [{'a': 2, 'b': 1}, {'a': 3, 'b': 2}, {'a': 1, 'b': 3}]
>>> results = order_by(items, ['b', 'a'])
>>> assert results == [{'a': 2, 'b': 1}, {'a': 3, 'b': 2}, {'a': 1, 'b': 3}]
↪ '': 2},
>>> results = order_by(items, ['a', 'b'])
>>> assert results == [{'a': 1, 'b': 3}, {'a': 2, 'b': 2}, {'a': 3, 'b': 1}]
↪ '': 1},
>>> results = order_by(items, ['-a', 'b'])
>>> assert results == [{'a': 3, 'b': 2}, {'a': 2, 'b': 1}, {'a': 1, 'b': 3}]
↪ '': 1},
>>> results = order_by(items, ['a', 'b'], [False, True])
>>> assert results == [{'a': 3, 'b': 2}, {'a': 1, 'b': 3}, {'a': 2, 'b': 1}]
↪ '': 1},
```

New in version 3.0.0.

Changed in version 3.2.0: Added *orders* argument.

Changed in version 3.2.0: Added `sort_by_order()` as alias.

Changed in version 4.0.0: Renamed from `order_by` to `order_by` and removed alias `sort_by_order`.

`pydash.collections.partition(collection, predicate=None)`

Creates an array of elements split into two groups, the first of which contains elements the *predicate* returns truthy for, while the second of which contains elements the *predicate* returns falsey for. The *predicate* is invoked with three arguments: (value, index|key, collection).

If a property name is provided for *predicate* the created *pluck()* style predicate returns the property value of the given element.

If an object is provided for *predicate* the created *where()* style predicate returns True for elements that have the properties of the given object, else False.

Parameters

- **collection** (*list|dict*) – Collection to iterate over.
- **predicate** (*mixed, optional*) – Predicate applied per iteration.

Returns List of grouped elements.

Return type list

Example

```
>>> partition([1, 2, 3, 4], lambda x: x >= 3)
[[3, 4], [1, 2]]
```

New in version 1.1.0.

`pydash.collections.pluck(collection, path)`

Retrieves the value of a specified property from all elements in the collection.

Parameters

- **collection** (*list*) – List of dicts.
- **path** (*str/list*) – Collection's path to pluck

Returns Plucked list.

Return type list

Example

```
>>> pluck([{'a': 1, 'b': 2}, {'a': 3, 'b': 4}, {'a': 5, 'b': 6}], 'a')
[1, 3, 5]
>>> pluck([[[0, 1]], [[2, 3]], [[4, 5]]], '0.1')
[1, 3, 5]
>>> pluck([{'a': {'b': 1}}, {'a': {'b': 2}}], 'a.b')
[1, 2]
>>> pluck([{'a': {'b': [0, 1]}}, {'a': {'b': [2, 3]}}], 'a.b.1')
[1, 3]
>>> pluck([{'a': {'b': [0, 1]}}, {'a': {'b': [2, 3]}}], ['a', 'b', 1])
[1, 3]
```

New in version 1.0.0.

Changed in version 4.0.0: Function removed.

Changed in version 4.0.1: Made property access deep.

`pydash.collections.reduce_` (*collection*, *iteratee=None*, *accumulator=None*)

Reduces a collection to a value which is the accumulated result of running each element in the collection through the iteratee, where each successive iteratee execution consumes the return value of the previous execution.

Parameters

- **collection** (*list/dict*) – Collection to iterate over.
- **iteratee** (*mixed*) – Iteratee applied per iteration.
- **accumulator** (*mixed, optional*) – Initial value of aggregator. Default is to use the result of the first iteration.

Returns Accumulator object containing results of reduction.

Return type mixed

Example

```
>>> reduce_([1, 2, 3, 4], lambda total, x: total * x)
24
```

New in version 1.0.0.

Changed in version 4.0.0: Removed aliases `foldl` and `inject`.

`pydash.collections.reduce_right` (*collection*, *iteratee=None*, *accumulator=None*)

This method is like `reduce_()` except that it iterates over elements of a *collection* from right to left.

Parameters

- **collection** (*list/dict*) – Collection to iterate over.
- **iteratee** (*mixed*) – Iteratee applied per iteration.

- **accumulator** (*mixed, optional*) – Initial value of aggregator. Default is to use the result of the first iteration.

Returns Accumulator object containing results of reduction.

Return type *mixed*

Example

```
>>> reduce_right([1, 2, 3, 4], lambda total, x: total ** x)
4096
```

New in version 1.0.0.

Changed in version 3.2.1: Fix bug where collection was not reversed correctly.

Changed in version 4.0.0: Removed alias `foldr`.

`pydash.collections.reductions` (*collection*, *iteratee=None*, *accumulator=None*, *from_right=False*)

This function is like `reduce_()` except that it returns a list of every intermediate value in the reduction operation.

Parameters

- **collection** (*list/dict*) – Collection to iterate over.
- **iteratee** (*mixed*) – Iteratee applied per iteration.
- **accumulator** (*mixed, optional*) – Initial value of aggregator. Default is to use the result of the first iteration.

Returns Results of each reduction operation.

Return type *list*

Example

```
>>> reductions([1, 2, 3, 4], lambda total, x: total * x)
[2, 6, 24]
```

Note: The last element of the returned list would be the result of using `reduce_()`.

New in version 2.0.0.

`pydash.collections.reductions_right` (*collection*, *iteratee=None*, *accumulator=None*)

This method is like `reductions()` except that it iterates over elements of a *collection* from right to left.

Parameters

- **collection** (*list/dict*) – Collection to iterate over.
- **iteratee** (*mixed*) – Iteratee applied per iteration.
- **accumulator** (*mixed, optional*) – Initial value of aggregator. Default is to use the result of the first iteration.

Returns Results of each reduction operation.

Return type list

Example

```
>>> reductions_right([1, 2, 3, 4], lambda total, x: total ** x)
[64, 4096, 4096]
```

Note: The last element of the returned list would be the result of using `reduce_()`.

New in version 2.0.0.

`pydash.collections.reject` (*collection*, *predicate=None*)

The opposite of `filter_()` this method returns the elements of a collection that the predicate does **not** return truthy for.

Parameters

- **collection** (*list*/*dict*) – Collection to iterate over.
- **predicate** (*mixed*, *optional*) – Predicate applied per iteration.

Returns Rejected elements of *collection*.

Return type list

Example

```
>>> reject([1, 2, 3, 4], lambda x: x >= 3)
[1, 2]
>>> reject([{'a': 0}, {'a': 1}, {'a': 2}], 'a')
[{'a': 0}]
>>> reject([{'a': 0}, {'a': 1}, {'a': 2}], {'a': 1})
[{'a': 0}, {'a': 2}]
```

New in version 1.0.0.

`pydash.collections.sample` (*collection*)

Retrieves a random element from a given *collection*.

Parameters **collection** (*list*/*dict*) – Collection to iterate over.

Returns Random element from the given collection.

Return type mixed

Example

```
>>> items = [1, 2, 3, 4, 5]
>>> results = sample(items)
>>> assert results in items
```

New in version 1.0.0.

Changed in version 4.0.0: Moved multiple samples functionality to `sample_size()`. This function now only returns a single random sample.

`pydash.collections.sample_size(collection, n=None)`

Retrieves list of *n* random elements from a collection.

Parameters

- **collection** (*list* / *dict*) – Collection to iterate over.
- **n** (*int*, *optional*) – Number of random samples to return.

Returns List of *n* sampled collection values.

Return type list

Examples

```
>>> items = [1, 2, 3, 4, 5]
>>> results = sample_size(items, 2)
>>> assert len(results) == 2
>>> assert set(items).intersection(results) == set(results)
```

New in version 4.0.0.

`pydash.collections.shuffle(collection)`

Creates a list of shuffled values, using a version of the Fisher-Yates shuffle.

Parameters **collection** (*list* / *dict*) – Collection to iterate over.

Returns Shuffled list of values.

Return type list

Example

```
>>> items = [1, 2, 3, 4]
>>> results = shuffle(items)
>>> assert len(results) == len(items)
>>> assert set(results) == set(items)
```

New in version 1.0.0.

`pydash.collections.size(collection)`

Gets the size of the *collection* by returning *len(collection)* for iterable objects.

Parameters **collection** (*list* / *dict*) – Collection to iterate over.

Returns Collection length.

Return type int

Example

```
>>> size([1, 2, 3, 4])
4
```

New in version 1.0.0.

`pydash.collections.some(collection, predicate=None)`

Checks if the predicate returns a truthy value for any element of a collection. The predicate is invoked with three arguments: (value, index|key, collection). If a property name is passed for predicate, the created `map_()` style predicate will return the property value of the given element. If an object is passed for predicate, the created `where()` style predicate will return `True` for elements that have the properties of the given object, else `False`.

Parameters

- **collection** (*list*/*dict*) – Collection to iterate over.
- **predicate** (*mixed*, *optional*) – Predicate applied per iteration.

Returns Whether any of the elements are truthy.

Return type `bool`

Example

```
>>> some([False, True, 0])
True
>>> some([False, 0, None])
False
>>> some([1, 2, 3, 4], lambda x: x >= 3)
True
>>> some([1, 2, 3, 4], lambda x: x == 0)
False
```

New in version 1.0.0.

Changed in version 4.0.0: Removed alias `any_`.

`pydash.collections.sort_by(collection, iteratee=None, reverse=False)`

Creates a list of elements, sorted in ascending order by the results of running each element in a *collection* through the *iteratee*.

Parameters

- **collection** (*list*/*dict*) – Collection to iterate over.
- **iteratee** (*mixed*, *optional*) – Iteratee applied per iteration.
- **reverse** (*bool*, *optional*) – Whether to reverse the sort. Defaults to `False`.

Returns Sorted list.

Return type `list`

Example

```
>>> sort_by({'a': 2, 'b': 3, 'c': 1})
[1, 2, 3]
>>> sort_by({'a': 2, 'b': 3, 'c': 1}, reverse=True)
[3, 2, 1]
>>> sort_by([{'a': 2}, {'a': 3}, {'a': 1}], 'a')
[{'a': 1}, {'a': 2}, {'a': 3}]
```

New in version 1.0.0.

`pydash.collections.to_list(collection)`

Converts the collection to a list.

Parameters `collection` (*list*/*dict*) – Collection to iterate over.

Returns Collection converted to list.

Return type list

Example

```
>>> results = to_list({'a': 1, 'b': 2, 'c': 3})
>>> assert set(results) == set([1, 2, 3])
>>> to_list((1, 2, 3, 4))
[1, 2, 3, 4]
```

New in version 1.0.0.

Functions

Functions that wrap other functions.

New in version 1.0.0.

`pydash.functions.after(func, n)`

Creates a function that executes *func*, with the arguments of the created function, only after being called *n* times.

Parameters

- **func** (*function*) – Function to execute.
- **n** (*int*) – Number of times *func* must be called before it is executed.

Returns Function wrapped in an `After` context.

Return type `After`

Example

```
>>> func = lambda a, b, c: (a, b, c)
>>> after_func = after(func, 3)
>>> after_func(1, 2, 3)
>>> after_func(1, 2, 3)
>>> after_func(1, 2, 3)
(1, 2, 3)
>>> after_func(4, 5, 6)
(4, 5, 6)
```

New in version 1.0.0.

Changed in version 3.0.0: Reordered arguments to make *func* first.

`pydash.functions.ary(func, n)`

Creates a function that accepts up to *n* arguments ignoring any additional arguments. Only positional arguments are capped. All keyword arguments are allowed through.

Parameters

- **func** (*function*) – Function to cap arguments for.
- **n** (*int*) – Number of arguments to accept.

Returns Function wrapped in an `Ary` context.

Return type `Ary`

Example

```
>>> func = lambda a, b, c=0, d=5: (a, b, c, d)
>>> ary_func = ary(func, 2)
>>> ary_func(1, 2, 3, 4, 5, 6)
(1, 2, 0, 5)
>>> ary_func(1, 2, 3, 4, 5, 6, c=10, d=20)
(1, 2, 10, 20)
```

New in version 3.0.0.

`pydash.functions.before` (*func*, *n*)

Creates a function that executes *func*, with the arguments of the created function, until it has been called *n* times.

Parameters

- **func** (*function*) – Function to execute.
- **n** (*int*) – Number of times *func* may be executed.

Returns Function wrapped in an `Before` context.

Return type `Before`

Example

```
>>> func = lambda a, b, c: (a, b, c)
>>> before_func = before(func, 3)
>>> before_func(1, 2, 3)
(1, 2, 3)
>>> before_func(1, 2, 3)
(1, 2, 3)
>>> before_func(1, 2, 3)
(1, 2, 3)
>>> before_func(1, 2, 3)
(1, 2, 3)
```

New in version 1.1.0.

Changed in version 3.0.0: Reordered arguments to make *func* first.

`pydash.functions.conjoin` (**funcs*)

Creates a function that composes multiple predicate functions into a single predicate that tests whether **all** elements of an object pass each predicate.

Parameters ***funcs** (*function*) – Function(s) to conjoin.

Returns Function(s) wrapped in a `Conjoin` context.

Return type `Conjoin`

Example

```
>>> conjoiner = conjoin(lambda x: isinstance(x, int), lambda x: x > 3)
>>> conjoiner([1, 2, 3])
False
>>> conjoiner([1.0, 2, 1])
False
>>> conjoiner([4.0, 5, 6])
False
>>> conjoiner([4, 5, 6])
True
```

New in version 2.0.0.

`pydash.functions.curry` (*func*, *arity=None*)

Creates a function that accepts one or more arguments of *func* that when invoked either executes *func* returning its result (if all *func* arguments have been provided) or returns a function that accepts one or more of the remaining *func* arguments, and so on.

Parameters

- **func** (*function*) – Function to curry.
- **arity** (*int*, *optional*) – Number of function arguments that can be accepted by curried function. Default is to use the number of arguments that are accepted by *func*.

Returns Function wrapped in a `Curry` context.

Return type `Curry`

Example

```
>>> func = lambda a, b, c: (a, b, c)
>>> currier = curry(func)
>>> currier = currier(1)
>>> assert isinstance(currier, Curry)
>>> currier = currier(2)
>>> assert isinstance(currier, Curry)
>>> currier = currier(3)
>>> currier
(1, 2, 3)
```

New in version 1.0.0.

`pydash.functions.curry_right` (*func*, *arity=None*)

This method is like `curry()` except that arguments are applied to *func* in the manner of `partial_right()` instead of `partial()`.

Parameters

- **func** (*function*) – Function to curry.
- **arity** (*int*, *optional*) – Number of function arguments that can be accepted by curried function. Default is to use the number of arguments that are accepted by *func*.

Returns Function wrapped in a `CurryRight` context.

Return type `CurryRight`

Example

```
>>> func = lambda a, b, c: (a, b, c)
>>> currier = curry_right(func)
>>> currier = currier(1)
>>> assert isinstance(currier, CurryRight)
>>> currier = currier(2)
>>> assert isinstance(currier, CurryRight)
>>> currier = currier(3)
>>> currier
(3, 2, 1)
```

New in version 1.1.0.

`pydash.functions.debounce` (*func*, *wait*, *max_wait=False*)

Creates a function that will delay the execution of *func* until after *wait* milliseconds have elapsed since the last time it was invoked. Subsequent calls to the debounced function will return the result of the last *func* call.

Parameters

- **func** (*function*) – Function to execute.
- **wait** (*int*) – Milliseconds to wait before executing *func*.
- **max_wait** (*optional*) – Maximum time to wait before executing *func*.

Returns Function wrapped in a `Debounce` context.

Return type `Debounce`

New in version 1.0.0.

`pydash.functions.delay` (*func*, *wait*, **args*, ***kwargs*)

Executes the *func* function after *wait* milliseconds. Additional arguments will be provided to *func* when it is invoked.

Parameters

- **func** (*function*) – Function to execute.
- **wait** (*int*) – Milliseconds to wait before executing *func*.
- ***args** (*optional*) – Arguments to pass to *func*.
- ****kwargs** (*optional*) – Keyword arguments to pass to *func*.

Returns Return from *func*.

Return type `mixed`

New in version 1.0.0.

`pydash.functions.disjoin` (**funcs*)

Creates a function that composes multiple predicate functions into a single predicate that tests whether **any** elements of an object pass each predicate.

Parameters ***funcs** (*function*) – Function(s) to disjoin.

Returns Function(s) wrapped in a `Disjoin` context.

Return type `Disjoin`

Example

```
>>> disjoiner = disjoin(lambda x: isinstance(x, float),
↳ lambda x: isinstance(x, int))
>>> disjoiner([1, '2', '3'])
True
>>> disjoiner([1.0, '2', '3'])
True
>>> disjoiner(['1', '2', '3'])
False
```

New in version 2.0.0.

`pydash.functions.flip(func)`

Creates a function that invokes the method with arguments reversed.

Parameters `func` (*function*) – Function to flip arguments for.

Returns Function wrapped in a `Flip` context.

Return type function

Example

```
>>> flipped = flip(lambda *args: args)
>>> flipped(1, 2, 3, 4)
(4, 3, 2, 1)
>>> flipped = flip(lambda *args: [i * 2 for i in args])
>>> flipped(1, 2, 3, 4)
[8, 6, 4, 2]
```

New in version 4.0.0.

`pydash.functions.flow(*funcs)`

Creates a function that is the composition of the provided functions, where each successive invocation is supplied the return value of the previous. For example, composing the functions `f()`, `g()`, and `h()` produces `h(g(f()))`.

Parameters `*funcs` (*function*) – Function(s) to compose.

Returns Function(s) wrapped in a `Flow` context.

Return type Flow

Example

```
>>> mult_5 = lambda x: x * 5
>>> div_10 = lambda x: x / 10.0
>>> pow_2 = lambda x: x ** 2
>>> ops = flow(sum, mult_5, div_10, pow_2)
>>> ops([1, 2, 3, 4])
25.0
```

New in version 2.0.0.

Changed in version 2.3.1: Added `pipe()` as alias.

Changed in version 4.0.0: Removed alias `pipe`.

`pydash.functions.flow_right(*funcs)`

This function is like `flow()` except that it creates a function that invokes the provided functions from right to left. For example, composing the functions `f()`, `g()`, and `h()` produces `f(g(h()))`.

Parameters `*funcs` (*function*) – Function(s) to compose.

Returns Function(s) wrapped in a `Flow` context.

Return type `Flow`

Example

```
>>> mult_5 = lambda x: x * 5
>>> div_10 = lambda x: x / 10.0
>>> pow_2 = lambda x: x ** 2
>>> ops = flow_right(mult_5, div_10, pow_2, sum)
>>> ops([1, 2, 3, 4])
50.0
```

New in version 1.0.0.

Changed in version 2.0.0: Added `flow_right()` and made `compose()` an alias.

Changed in version 2.3.1: Added `pipe_right()` as alias.

Changed in version 4.0.0: Removed aliases `pipe_right` and `compose`.

`pydash.functions.iterated(func)`

Creates a function that is composed with itself. Each call to the iterated function uses the previous function call's result as input. Returned `Iterated` instance can be called with (*initial*, *n*) where *initial* is the initial value to seed *func* with and *n* is the number of times to call *func*.

Parameters `func` (*function*) – Function to iterate.

Returns Function wrapped in a `Iterated` context.

Return type `Iterated`

Example

```
>>> doubler = iterated(lambda x: x * 2)
>>> doubler(4, 5)
128
>>> doubler(3, 9)
1536
```

New in version 2.0.0.

`pydash.functions.juxtapose(*funcs)`

Creates a function whose return value is a list of the results of calling each *funcs* with the supplied arguments.

Parameters `*funcs` (*function*) – Function(s) to juxtapose.

Returns Function wrapped in a `Juxtapose` context.

Return type `Juxtapose`

Example

```
>>> double = lambda x: x * 2
>>> triple = lambda x: x * 3
>>> quadruple = lambda x: x * 4
>>> juxtapose(double, triple, quadruple)(5)
[10, 15, 20]
```

New in version 2.0.0.

`pydash.functions.negate(func)`

Creates a function that negates the result of the predicate *func*. The *func* function is executed with the arguments of the created function.

Parameters *func* (*function*) – Function to negate execute.

Returns Function wrapped in a `Negate` context.

Return type `Negate`

Example

```
>>> not_is_number = negate(lambda x: isinstance(x, (int, float)))
>>> not_is_number(1)
False
>>> not_is_number('1')
True
```

New in version 1.1.0.

`pydash.functions.once(func)`

Creates a function that is restricted to execute *func* once. Repeat calls to the function will return the value of the first call.

Parameters *func* (*function*) – Function to execute.

Returns Function wrapped in a `Once` context.

Return type `Once`

Example

```
>>> oncer = once(lambda *args: args[0])
>>> oncer(5)
5
>>> oncer(6)
5
```

New in version 1.0.0.

`pydash.functions.over_args(func, *transforms)`

Creates a function that runs each argument through a corresponding transform function.

Parameters

- **func** (*function*) – Function to wrap.

- ***transforms** (*function*) – Functions to transform arguments, specified as individual functions or lists of functions.

Returns Function wrapped in a `OverArgs` context.

Return type `OverArgs`

Example

```
>>> squared = lambda x: x ** 2
>>> double = lambda x: x * 2
>>> modder = over_args(lambda x, y: [x, y], squared, double)
>>> modder(5, 10)
[25, 20]
```

New in version 3.3.0.

Changed in version 4.0.0: Renamed from `mod_args` to `over_args`.

`pydash.functions.partial` (*func*, **args*, ***kargs*)

Creates a function that, when called, invokes *func* with any additional partial arguments prepended to those provided to the new function.

Parameters

- **func** (*function*) – Function to execute.
- ***args** (*optional*) – Partial arguments to prepend to function call.
- ****kargs** (*optional*) – Partial keyword arguments to bind to function call.

Returns Function wrapped in a `Partial` context.

Return type `Partial`

Example

```
>>> dropper = partial(lambda array, n: array[n:], [1, 2, 3, 4])
>>> dropper(2)
[3, 4]
>>> dropper(1)
[2, 3, 4]
>>> myrest = partial(lambda array, n: array[n:], n=1)
>>> myrest([1, 2, 3, 4])
[2, 3, 4]
```

New in version 1.0.0.

`pydash.functions.partial_right` (*func*, **args*, ***kargs*)

This method is like `partial()` except that partial arguments are appended to those provided to the new function.

Parameters

- **func** (*function*) – Function to execute.
- ***args** (*optional*) – Partial arguments to append to function call.
- ****kargs** (*optional*) – Partial keyword arguments to bind to function call.

Returns Function wrapped in a `Partial` context.

Return type `Partial`

Example

```
>>> myrest = partial_right(lambda array, n: array[n:], 1)
>>> myrest([1, 2, 3, 4])
[2, 3, 4]
```

New in version 1.0.0.

`pydash.functions.rearg` (*func*, **indexes*)

Creates a function that invokes *func* with arguments arranged according to the specified indexes where the argument value at the first index is provided as the first argument, the argument value at the second index is provided as the second argument, and so on.

Parameters

- **func** (*function*) – Function to rearrange arguments for.
- ***indexes** (*int*) – The arranged argument indexes.

Returns Function wrapped in a `Rearg` context.

Return type `Rearg`

Example

```
>>> jumble = rearg(lambda *args: args, 1, 2, 3)
>>> jumble(1, 2, 3)
(2, 3, 1)
>>> jumble('a', 'b', 'c', 'd', 'e')
('b', 'c', 'd', 'a', 'e')
```

New in version 3.0.0.

`pydash.functions.spread` (*func*)

Creates a function that invokes *func* with the array of arguments provided to the created function.

Parameters **func** (*function*) – Function to spread.

Returns Function wrapped in a `Spread` context.

Return type `Spread`

Example

```
>>> greet = spread(lambda people: 'Hello ' + ', '.join(people) + '!')
>>> greet(['Mike', 'Don', 'Leo'])
'Hello Mike, Don, Leo!'
```

New in version 3.1.0.

`pydash.functions.throttle` (*func*, *wait*)

Creates a function that, when executed, will only call the *func* function at most once per every *wait* milliseconds. Subsequent calls to the throttled function will return the result of the last *func* call.

Parameters

- **func** (*function*) – Function to throttle.
- **wait** (*int*) – Milliseconds to wait before calling *func* again.

Returns Results of last *func* call.

Return type mixed

New in version 1.0.0.

`pydash.functions.unary(func)`

Creates a function that accepts up to one argument, ignoring any additional arguments.

Parameters **func** (*function*) – Function to cap arguments for.

Returns Function wrapped in an `Ary` context.

Return type `Ary`

Example

```
>>> func = lambda a, b=1, c=0, d=5: (a, b, c, d)
>>> unary_func = unary(func)
>>> unary_func(1, 2, 3, 4, 5, 6)
(1, 1, 0, 5)
>>> unary_func(1, 2, 3, 4, 5, 6, b=0, c=10, d=20)
(1, 0, 10, 20)
```

New in version 4.0.0.

`pydash.functions.wrap(value, func)`

Creates a function that provides *value* to the wrapper function as its first argument. Additional arguments provided to the function are appended to those provided to the wrapper function.

Parameters

- **value** (*mixed*) – Value provided as first argument to function call.
- **func** (*function*) – Function to execute.

Returns Function wrapped in a `Partial` context.

Return type `Partial`

Example

```
>>> wrapper = wrap('hello', lambda *args: args)
>>> wrapper(1, 2)
('hello', 1, 2)
```

New in version 1.0.0.

Numerical

Numerical/mathemetical related functions.

New in version 2.1.0.

`pydash.numerical.add(a, b)`

Adds two numbers.

Parameters

- **a** (*number*) – First number to add.
- **b** (*number*) – Second number to add.

Returns number

Example

```
>>> add(10, 5)
15
```

New in version 2.1.0.

Changed in version 3.3.0: Support adding two numbers when passed as positional arguments.

Changed in version 4.0.0: Only support two argument addition.

`pydash.numerical.ceil(x, precision=0)`

Round number up to precision.

Parameters

- **x** (*number*) – Number to round up.
- **precision** (*int, optional*) – Rounding precision. Defaults to 0.

Returns Number rounded up.

Return type int

Example

```
>>> ceil(3.275) == 4.0
True
>>> ceil(3.215, 1) == 3.3
True
>>> ceil(6.004, 2) == 6.01
True
```

New in version 3.3.0.

`pydash.numerical.clamp(x, lower, upper=None)`

Clamps number within the inclusive lower and upper bounds.

Parameters

- **x** (*number*) – Number to clamp.
- **lower** (*number, optional*) – Lower bound.
- **upper** (*number*) – Upper bound

Returns number

Example

```
>>> clamp(-10, -5, 5)
-5
>>> clamp(10, -5, 5)
5
>>> clamp(10, 5)
5
>>> clamp(-10, 5)
-10
```

New in version 4.0.0.

`pydash.numerical.divide(dividend, divisor)`

Divide two numbers.

Parameters

- **dividend** (*int/float*) – The first number in a division.
- **divisor** (*int/float*) – The second number in a division.

Returns Returns the quotient.

Return type `int/float`

Example

```
>>> divide(20, 5)
4.0
>>> divide(1.5, 3)
0.5
>>> divide(None, None)
1.0
>>> divide(5, None)
5.0
```

New in version 4.0.0.

`pydash.numerical.floor(x, precision=0)`

Round number down to precision.

Parameters

- **x** (*number*) – Number to round down.
- **precision** (*int, optional*) – Rounding precision. Defaults to 0.

Returns Number rounded down.

Return type `int`

Example

```
>>> floor(3.75) == 3.0
True
>>> floor(3.215, 1) == 3.2
True
```

```
>>> floor(0.046, 2) == 0.04
True
```

New in version 3.3.0.

`pydash.numerical.max_(collection, default=<pydash.helpers._NoValue object>)`

Retrieves the maximum value of a *collection*.

Parameters

- **collection** (*list* | *dict*) – Collection to iterate over.
- **default** (*mixed*, *optional*) – Value to return if *collection* is empty.

Returns Maximum value.

Return type `mixed`

Example

```
>>> max_([1, 2, 3, 4])
4
>>> max_([], default=-1)
-1
```

New in version 1.0.0.

Changed in version 4.0.0: Moved iteratee support to `max_by()`.

`pydash.numerical.max_by(collection, iteratee=None, default=<pydash.helpers._NoValue object>)`

Retrieves the maximum value of a *collection*.

Parameters

- **collection** (*list* | *dict*) – Collection to iterate over.
- **iteratee** (*mixed*, *optional*) – Iteratee applied per iteration.
- **default** (*mixed*, *optional*) – Value to return if *collection* is empty.

Returns Maximum value.

Return type `mixed`

Example

```
>>> max_by([1.0, 1.5, 1.8], math.floor)
1.0
>>> max_by([{'a': 1}, {'a': 2}, {'a': 3}], 'a')
{'a': 3}
>>> max_by([], default=-1)
-1
```

New in version 4.0.0.

`pydash.numerical.mean(collection)`

Calculate arithmetic mean of each element in *collection*.

Parameters **collection** (*list* | *dict*) – Collection to process.

Returns Result of mean.

Return type float

Example

```
>>> mean([1, 2, 3, 4])
2.5
```

New in version 2.1.0.

Changed in version 4.0.0: Removed `average` and `avg` aliases. Moved iteratee functionality to `mean_by()`.

`pydash.numerical.mean_by(collection, iteratee=None)`

Calculate arithmetic mean of each element in *collection*. If *iteratee* is passed, each element of *collection* is passed through a *iteratee* before the mean is computed.

Parameters

- **collection** (*list* | *dict*) – Collection to process.
- **iteratee** (*mixed*, *optional*) – Iteratee applied per iteration.

Returns Result of mean.

Return type float

Example

```
>>> mean_by([1, 2, 3, 4], lambda x: x ** 2)
7.5
```

New in version 4.0.0.

`pydash.numerical.median(collection, iteratee=None)`

Calculate median of each element in *collection*. If *iteratee* is passed, each element of *collection* is passed through a *iteratee* before the median is computed.

Parameters

- **collection** (*list* | *dict*) – Collection to process.
- **iteratee** (*mixed*, *optional*) – Iteratee applied per iteration.

Returns Result of median.

Return type float

Example

```
>>> median([1, 2, 3, 4, 5])
3
>>> median([1, 2, 3, 4])
2.5
```

New in version 2.1.0.

`pydash.numerical.min_(collection, default=<pydash.helpers._NoValue object>)`

Retrieves the minimum value of a *collection*.

Parameters

- **collection** (*list* / *dict*) – Collection to iterate over.
- **default** (*mixed*, *optional*) – Value to return if *collection* is empty.

Returns Minimum value.

Return type `mixed`

Example

```
>>> min_([1, 2, 3, 4])
1
>>> min_([], default=100)
100
```

New in version 1.0.0.

Changed in version 4.0.0: Moved iteratee support to `min_by()`.

`pydash.numerical.min_by(collection, iteratee=None, default=<pydash.helpers._NoValue object>)`

Retrieves the minimum value of a *collection*.

Parameters

- **collection** (*list* / *dict*) – Collection to iterate over.
- **iteratee** (*mixed*, *optional*) – Iteratee applied per iteration.
- **default** (*mixed*, *optional*) – Value to return if *collection* is empty.

Returns Minimum value.

Return type `mixed`

Example

```
>>> min_by([1.8, 1.5, 1.0], math.floor)
1.8
>>> min_by([{'a': 1}, {'a': 2}, {'a': 3}], 'a')
{'a': 1}
>>> min_by([], default=100)
100
```

New in version 4.0.0.

`pydash.numerical.moving_mean(array, size)`

Calculate moving mean of each element of *array*.

Parameters

- **array** (*list*) – List to process.
- **size** (*int*) – Window size.

Returns Result of moving average.

Return type `list`

Example

```
>>> moving_mean(range(10), 1)
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
>>> moving_mean(range(10), 5)
[2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
>>> moving_mean(range(10), 10)
[4.5]
```

New in version 2.1.0.

Changed in version 4.0.0: Rename to `moving_mean` and remove `moving_average` and `moving_avg` aliases.

`pydash.numerical.multiply(multiplier, multiplicand)`

Multiply two numbers.

Parameters

- **multiplier** (*int/float*) – The first number in a multiplication.
- **multiplicand** (*int/float*) – The second number in a multiplication.

Returns Returns the product.

Return type `int/float`

Example

```
>>> multiply(4, 5)
20
>>> multiply(10, 4)
40
>>> multiply(None, 10)
10
>>> multiply(None, None)
1
```

New in version 4.0.0.

`pydash.numerical.power(x, n)`

Calculate exponentiation of x raised to the n power.

Parameters

- **x** (*number*) – Base number.
- **n** (*number*) – Exponent.

Returns Result of calculation.

Return type `number`

Example

```
>>> power(5, 2)
25
>>> power(12.5, 3)
1953.125
```

New in version 2.1.0.

Changed in version 4.0.0: Removed alias `pow_`.

`pydash.numerical.round_(x, precision=0)`

Round number to precision.

Parameters

- **x** (*number*) – Number to round.
- **precision** (*int*, *optional*) – Rounding precision. Defaults to 0.

Returns Rounded number.

Return type `int`

Example

```
>>> round_(3.275) == 3.0
True
>>> round_(3.275, 1) == 3.3
True
```

New in version 2.1.0.

Changed in version 4.0.0: Remove alias `curve`.

`pydash.numerical.scale(array, maximum=1)`

Scale list of value to a maximum number.

Parameters

- **array** (*list*) – Numbers to scale.
- **maximum** (*number*) – Maximum scale value.

Returns Scaled numbers.

Return type `list`

Example

```
>>> scale([1, 2, 3, 4])
[0.25, 0.5, 0.75, 1.0]
>>> scale([1, 2, 3, 4], 1)
[0.25, 0.5, 0.75, 1.0]
>>> scale([1, 2, 3, 4], 4)
[1.0, 2.0, 3.0, 4.0]
>>> scale([1, 2, 3, 4], 2)
[0.5, 1.0, 1.5, 2.0]
```

New in version 2.1.0.

`pydash.numerical.slope(point1, point2)`

Calculate the slope between two points.

Parameters

- **point1** (*list/tuple*) – X and Y coordinates of first point.
- **point2** (*list/tuple*) – X and Y coordinates of second point.

Returns Calculated slope.

Return type float

Example

```
>>> slope((1, 2), (4, 8))
2.0
```

New in version 2.1.0.

`pydash.numerical.std_deviation` (*array*)

Calculate standard deviation of list of numbers.

Parameters **array** (*list*) – List to process.

Returns Calculated standard deviation.

Return type float

Example

```
>>> round(std_deviation([1, 18, 20, 4]), 2) == 8.35
True
```

New in version 2.1.0.

Changed in version 4.0.0: Remove alias `sigma`.

`pydash.numerical.sum_` (*collection*)

Sum each element in *collection*.

Parameters **collection** (*list|dict|number*) – Collection to process or first number to add.

Returns Result of summation.

Return type number

Example

```
>>> sum_([1, 2, 3, 4])
10
```

New in version 2.1.0.

Changed in version 3.3.0: Support adding two numbers when passed as positional arguments.

Changed in version 4.0.0: Move iteratee support to `sum_by()`. Move two argument addition to `add()`.

`pydash.numerical.sum_by` (*collection, iteratee=None*)

Sum each element in *collection*. If *iteratee* is passed, each element of *collection* is passed through a *iteratee* before the summation is computed.

Parameters

- **collection** (*list / dict / number*) – Collection to process or first number to add.
- **iteratee** (*mixed / number, optional*) – Iteratee applied per iteration or second number to add.

Returns Result of summation.

Return type number

Example

```
>>> sum_by([1, 2, 3, 4], lambda x: x ** 2)
30
```

New in version 4.0.0.

`pydash.numerical.subtract` (*minuend, subtrahend*)

Subtracts two numbers.

Parameters

- **minuend** (*int / float*) – Value passed in by the user.
- **subtrahend** (*int / float*) – Value passed in by the user.

Returns Result of the difference from the given values.

Return type int/float

Example

```
>>> subtract(10, 5)
5
>>> subtract(-10, 4)
-14
>>> subtract(2, 0.5)
1.5
```

New in version 4.0.0.

`pydash.numerical.transpose` (*array*)

Transpose the elements of *array*.

Parameters **array** (*list*) – List to process.

Returns Transposed list.

Return type list

Example

```
>>> transpose([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

New in version 2.1.0.

`pydash.numerical.variance` (*array*)

Calculate the variance of the elements in *array*.

Parameters `array` (*list*) – List to process.

Returns Calculated variance.

Return type float

Example

```
>>> variance([1, 18, 20, 4])
69.6875
```

New in version 2.1.0.

`pydash.numerical.zscore` (*collection*, *iteratee=None*)

Calculate the standard score assuming normal distribution. If *iteratee* is passed, each element of *collection* is passed through a *iteratee* before the standard score is computed.

Parameters

- **collection** (*list* | *dict*) – Collection to process.
- **iteratee** (*mixed*, *optional*) – Iteratee applied per iteration.

Returns Calculated standard score.

Return type float

Example

```
>>> results = zscore([1, 2, 3])
```

```
# [-1.224744871391589, 0.0, 1.224744871391589]
```

New in version 2.1.0.

Objects

Functions that operate on lists, dicts, and other objects.

New in version 1.0.0.

`pydash.objects.assign` (*obj*, **sources*)

Assigns properties of source object(s) to the destination object.

Parameters

- **obj** (*dict*) – Destination object whose properties will be modified.
- **sources** (*dict*) – Source objects to assign to *obj*.

Returns Modified *obj*.

Return type dict

Warning: *obj* is modified in place.

Example

```
>>> obj = {}
>>> obj2 = assign(obj, {'a': 1}, {'b': 2}, {'c': 3})
>>> obj == {'a': 1, 'b': 2, 'c': 3}
True
>>> obj is obj2
True
```

New in version 1.0.0.

Changed in version 2.3.2: Apply `clone_deep()` to each *source* before assigning to *obj*.

Changed in version 3.0.0: Allow iteratees to accept partial arguments.

Changed in version 3.4.4: Shallow copy each *source* instead of deep copying.

Changed in version 4.0.0: Moved *iteratee* argument to `assign_with()`. Removed alias `extend`.

`pydash.objects.assign_with(obj, *sources, **kargs)`

This method is like `assign()` except that it accepts *customizer* which is invoked to produce the assigned values. If *customizer* returns `None`, assignment is handled by the method instead. The *customizer* is invoked with five arguments: (*obj_value*, *src_value*, *key*, *obj*, *source*).

Parameters

- **obj** (*dict*) – Destination object whose properties will be modified.
- **sources** (*dict*) – Source objects to assign to *obj*.

Keyword Arguments **customizer** (*mixed*, *optional*) – Customizer applied per iteration.

Returns Modified *obj*.

Return type `dict`

Warning: *obj* is modified in place.

Example

```
>>> customizer = lambda o, s: s if o is None else o
>>> results = assign({'a': 1}, {'b': 2}, {'a': 3}, customizer)
>>> results == {'a': 1, 'b': 2}
True
```

New in version 4.0.0.

`pydash.objects.callables(obj)`

Creates a sorted list of keys of an object that are callable.

Parameters **obj** (*list*/*dict*) – Object to inspect.

Returns All keys whose values are callable.

Return type `list`

Example

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

New in version 1.0.0.

Changed in version 2.0.0: Renamed functions to callables.

Changed in version 4.0.0: Removed alias methods.

`pydash.objects.clone(value)`

Creates a clone of *value*.

Parameters *value* (*list/dict*) – Object to clone.

Example

```
>>> x = {'a': 1, 'b': 2, 'c': {'d': 3}}
>>> y = clone(x)
>>> y == y
True
>>> x is y
False
>>> x['c'] is y['c']
True
```

Returns Cloned object.

Return type listdict

New in version 1.0.0.

Changed in version 4.0.0: Moved ‘iteratee’ parameter to `clone_with()`.

`pydash.objects.clone_deep(value)`

Creates a deep clone of *value*. If a iteratee is provided it will be executed to produce the cloned values.

Parameters *value* (*list/dict*) – Object to clone.

Returns Cloned object.

Return type listdict

Example

```
>>> x = {'a': 1, 'b': 2, 'c': {'d': 3}}
>>> y = clone_deep(x)
>>> y == y
True
>>> x is y
False
>>> x['c'] is y['c']
False
```

New in version 1.0.0.

Changed in version 4.0.0: Moved ‘iteratee’ parameter to `clone_deep_with()`.

`pydash.objects.clone_deep_with(value, customizer=None)`

This method is like `clone_with()` except that it recursively clones *value*.

Parameters

- **value** (*list/dict*) – Object to clone.
- **customizer** (*callable, optional*) – Function to customize cloning.

Returns Cloned object.

Return type listdict

`pydash.objects.clone_with(value, customizer=None)`

This method is like `clone()` except that it accepts *customizer* which is invoked to produce the cloned value. If *customizer* returns `None`, cloning is handled by the method instead. The *customizer* is invoked with up to three arguments: (*value*, *index|key*, *object*).

Parameters

- **value** (*list/dict*) – Object to clone.
- **customizer** (*callable, optional*) – Function to customize cloning.

Returns Cloned object.

Return type listdict

Example

```
>>> x = {'a': 1, 'b': 2, 'c': {'d': 3}}
>>> cbk = lambda v, k: v + 2 if isinstance(v, int) and k else None
>>> y = clone_with(x, cbk)
>>> y == {'a': 3, 'b': 4, 'c': {'d': 3}}
True
```

`pydash.objects.defaults(obj, *sources)`

Assigns properties of source object(s) to the destination object for all destination properties that resolve to undefined.

Parameters

- **obj** (*dict*) – Destination object whose properties will be modified.
- **sources** (*dict*) – Source objects to assign to *obj*.

Returns Modified *obj*.

Return type dict

Warning: *obj* is modified in place.

Example

```
>>> obj = {'a': 1}
>>> obj2 = defaults(obj, {'b': 2}, {'c': 3}, {'a': 4})
>>> obj is obj2
True
>>> obj == {'a': 1, 'b': 2, 'c': 3}
True
```

New in version 1.0.0.

`pydash.objects.defaults_deep(obj, *sources)`

This method is like `defaults()` except that it recursively assigns default properties.

Parameters

- **obj** (*dict*) – Destination object whose properties will be modified.
- **sources** (*dict*) – Source objects to assign to *obj*.

Returns Modified *obj*.

Return type dict

Warning: *obj* is modified in place.

Example

```
>>> obj = {'a': {'b': 1}}
>>> obj2 = defaults_deep(obj, {'a': {'b': 2, 'c': 3}})
>>> obj is obj2
True
>>> obj == {'a': {'b': 1, 'c': 3}}
True
```

New in version 3.3.0.

`pydash.objects.find_key(obj, predicate=None)`

This method is like `pydash.arrays.find_index()` except that it returns the key of the first element that passes the predicate check, instead of the element itself.

Parameters

- **obj** (*list* | *dict*) – Object to search.
- **predicate** (*mixed*) – Predicate applied per iteration.

Returns Found key or None.

Return type mixed

Example

```
>>> find_key({'a': 1, 'b': 2, 'c': 3}, lambda x: x == 1)
'a'
>>> find_key([1, 2, 3, 4], lambda x: x == 1)
0
```

New in version 1.0.0.

`pydash.objects.find_last_key(obj, predicate=None)`

This method is like `find_key()` except that it iterates over elements of a collection in the opposite order.

Parameters

- **obj** (*list/dict*) – Object to search.
- **predicate** (*mixed*) – Predicate applied per iteration.

Returns Found key or None.

Return type `mixed`

Example

```
>>> find_last_key({'a': 1, 'b': 2, 'c': 3}, lambda x: x == 1)
'a'
>>> find_last_key([1, 2, 3, 1], lambda x: x == 1)
3
```

Changed in version 4.0.0: Made into its own function (instead of an alias of `find_key`) with proper reverse find implementation.

`pydash.objects.for_in(obj, iteratee=None)`

Iterates over own and inherited enumerable properties of *obj*, executing *iteratee* for each property.

Parameters

- **obj** (*list/dict*) – Object to process.
- **iteratee** (*mixed*) – Iteratee applied per iteration.

Returns *obj*.

Return type `listdict`

Example

```
>>> obj = {}
>>> def cb(v, k): obj[k] = v
>>> results = for_in({'a': 1, 'b': 2, 'c': 3}, cb)
>>> results == {'a': 1, 'b': 2, 'c': 3}
True
>>> obj == {'a': 1, 'b': 2, 'c': 3}
True
```

New in version 1.0.0.

Changed in version 4.0.0: Removed alias `for_own`.

`pydash.objects.for_in_right(obj, iteratee=None)`

This function is like `for_in()` except it iterates over the properties in reverse order.

Parameters

- **obj** (*list/dict*) – Object to process.

- **iteratee** (*mixed*) – Iteratee applied per iteration.

Returns *obj*.

Return type listdict

Example

```
>>> data = {'product': 1}
>>> def cb(v): data['product'] *= v
>>> for_in_right([1, 2, 3, 4], cb)
[1, 2, 3, 4]
>>> data['product'] == 24
True
```

New in version 1.0.0.

Changed in version 4.0.0: Removed alias `for_own_right`.

`pydash.objects.get` (*obj*, *path*, *default=None*)

Get the value at any depth of a nested object based on the path described by *path*. If path doesn't exist, *default* is returned.

Parameters

- **obj** (*list|dict*) – Object to process.
- **path** (*str|list*) – List or . delimited string of path describing path.

Keyword Arguments **default** (*mixed*) – Default value to return if path doesn't exist. Defaults to None.

Returns Value of *obj* at path.

Return type mixed

Example

```
>>> get({}, 'a.b.c') is None
True
>>> get({'a': {'b': {'c': [1, 2, 3, 4]}}}, 'a.b.c.1')
2
>>> get({'a': {'b': [0, {'c': [1, 2]}}}}, 'a.b.1.c.1')
2
>>> get({'a': {'b': [0, {'c': [1, 2]}}}}, 'a.b.1.c.2') is None
True
```

New in version 2.0.0.

Changed in version 2.2.0: Support escaping "." delimiter in single string path key.

Changed in version 3.3.0: Added `get()` as main definition and `get_path()` as alias. Made `deep_get()` an alias.

Changed in version 3.4.7: Fixed bug where an iterable default was iterated over instead of being returned when an object path wasn't found.

Changed in version 4.0.0: Support attribute access on *obj* if item access fails. Removed aliases `get_path` and `deep_get`.

`pydash.objects.has(obj, path)`
Checks if *path* exists as a key of *obj*.

Parameters

- **obj** (*mixed*) – Object to test.
- **path** (*mixed*) – Path to test for. Can be a list of nested keys or a . delimited string of path describing the path.

Returns Whether *obj* has *path*.

Return type bool

Example

```
>>> has([1, 2, 3], 1)
True
>>> has({'a': 1, 'b': 2}, 'b')
True
>>> has({'a': 1, 'b': 2}, 'c')
False
>>> has({'a': {'b': [0, {'c': [1, 2]}]}}, 'a.b.1.c.1')
True
>>> has({'a': {'b': [0, {'c': [1, 2]}]}}, 'a.b.1.c.2')
False
```

New in version 1.0.0.

Changed in version 3.0.0: Return `False` on `ValueError` when checking path.

Changed in version 3.3.0: Added `deep_has()` as alias. Added `has_path()` as alias.

Changed in version 4.0.0: Removed aliases `deep_has` and `has_path`.

`pydash.objects.invert(obj)`
Creates an object composed of the inverted keys and values of the given object.

Parameters

- **obj** (*dict*) – Dict to invert.
- **multivalue** (*bool, optional*) – Whether to return inverted values as lists. Defaults to `False`.

Returns Inverted dict.

Return type dict

Example

```
>>> results = invert({'a': 1, 'b': 2, 'c': 3})
>>> results == {1: 'a', 2: 'b', 3: 'c'}
True
```

Note: Assumes *obj* values are hashable as dict keys.

New in version 1.0.0.

Changed in version 2.0.0: Added `multivalue` argument.

Changed in version 4.0.0: Moved `multivalue=True` functionality to `invert_by()`.

`pydash.objects.invert_by(obj, iteratee=None)`

This method is like `invert()` except that the inverted object is generated from the results of running each element of object thru `iteratee`. The corresponding inverted value of each inverted key is a list of keys responsible for generating the inverted value. The `iteratee` is invoked with one argument: `(value)`.

Parameters `obj` (*dict*) – Object to invert.

Returns Inverted dict.

Return type dict

Example

```
>>> obj = {'a': 1, 'b': 2, 'c': 1}
>>> results = invert_by(obj) # {1: ['a', 'c'], 2: ['b']}
>>> set(results[1]) == set(['a', 'c'])
True
>>> set(results[2]) == set(['b'])
True
>>> results2 = invert_by(obj, lambda value: 'group' + str(value))
>>> results2['group1'] == results[1]
True
>>> results2['group2'] == results[2]
True
```

Note: Assumes `obj` values are hashable as dict keys.

New in version 4.0.0.

`pydash.objects.invoke(obj, path, *args, **kwargs)`

Invokes the method at path of object.

Parameters

- `obj` (*dict*) – The object to query.
- `path` (*list/str*) – The path of the method to invoke.
- `args` (*optional*) – Arguments to pass to method call.
- `kwargs` (*optional*) – Keyword arguments to pass to method call.

Returns Result of the invoked method.

Return type mixed

Example

```
>>> obj = {'a': [{'b': {'c': [1, 2, 3, 4]}]}}
>>> invoke(obj, 'a[0].b.c.pop', 1)
2
>>> obj
{'a': [{'b': {'c': [1, 3, 4]}]}}
```

New in version 1.0.0.

`pydash.objects.keys(obj)`

Creates a list composed of the keys of *obj*.

Parameters *obj* (*mixed*) – Object to extract keys from.

Returns List of keys.

Return type list

Example

```
>>> keys([1, 2, 3])
[0, 1, 2]
>>> set(keys({'a': 1, 'b': 2, 'c': 3})) == set(['a', 'b', 'c'])
True
```

New in version 1.0.0.

Changed in version 1.1.0: Added `keys_in` as alias.

Changed in version 4.0.0: Removed alias `keys_in`.

`pydash.objects.map_keys(obj, iteratee=None)`

The opposite of `map_values()`, this method creates an object with the same values as object and keys generated by running each own enumerable string keyed property of object thru *iteratee*. The *iteratee* is invoked with three arguments: (value, key, object).

Parameters

- *obj* (*list/dict*) – Object to map.
- *iteratee* (*mixed*) – Iteratee applied per iteration.

Returns Results of running *obj* through *iteratee*.

Return type listdict

Example

```
>>> callback = lambda value, key: key * 2
>>> results = map_keys({'a': 1, 'b': 2, 'c': 3}, callback)
>>> results == {'aa': 1, 'bb': 2, 'cc': 3}
True
```

New in version 3.3.0.

`pydash.objects.map_values(obj, iteratee=None)`

Creates an object with the same keys as object and values generated by running each string keyed property of object thru *iteratee*. The *iteratee* is invoked with three arguments: (value, key, object).

Parameters

- *obj* (*list/dict*) – Object to map.
- *iteratee* (*mixed*) – Iteratee applied per iteration.

Returns Results of running *obj* through *iteratee*.

Return type listdict

Example

```
>>> results = map_values({'a': 1, 'b': 2, 'c': 3}, lambda x: x * 2)
>>> results == {'a': 2, 'b': 4, 'c': 6}
True
>>> results = map_values({'a': 1, 'b': {'d': 4}, 'c': 3}, {'d': 4})
>>> results == {'a': False, 'b': True, 'c': False}
True
```

New in version 1.0.0.

`pydash.objects.map_values_deep(obj, iteratee=None, property_path=<pydash.helpers._NoValue object>)`

Map all non-object values in *obj* with return values from *iteratee*. The *iteratee* is invoked with two arguments: (*obj_value*, *property_path*) where *property_path* contains the list of path keys corresponding to the path of *obj_value*.

Parameters

- **obj** (*list/dict*) – Object to map.
- **iteratee** (*function*) – Iteratee applied to each value.

Returns The modified object.

Return type mixed

Warning: *obj* is modified in place.

Example

```
>>> x = {'a': 1, 'b': {'c': 2}}
>>> y = map_values_deep(x, lambda val: val * 2)
>>> y == {'a': 2, 'b': {'c': 4}}
True
>>> z = map_values_deep(x, lambda val, props: props)
>>> z == {'a': ['a'], 'b': {'c': ['b', 'c']}}
True
```

Changed in version 3.0.0: Allow iteratees to accept partial arguments.

Changed in version 4.0.0: Renamed from `deep_map_values` to `map_values_deep`.

`pydash.objects.merge(obj, *sources)`

Recursively merges properties of the source object(s) into the destination object. Subsequent sources will overwrite property assignments of previous sources.

Parameters

- **obj** (*dict*) – Destination object to merge source(s) into.
- **sources** (*dict*) – Source objects to merge from. subsequent sources overwrite previous ones.

Returns Merged object.

Return type dict

Warning: *obj* is modified in place.

Example

```
>>> obj = {'a': 2}
>>> obj2 = merge(obj, {'a': 1}, {'b': 2, 'c': 3}, {'d': 4})
>>> obj2 == {'a': 1, 'b': 2, 'c': 3, 'd': 4}
True
>>> obj is obj2
True
```

New in version 1.0.0.

Changed in version 2.3.2: Apply *clone_deep()* to each *source* before assigning to *obj*.

Changed in version 2.3.2: Allow *iteratee* to be passed by reference if it is the last positional argument.

Changed in version 4.0.0: Moved *iteratee* argument to *merge_with()*.

`pydash.objects.merge_with(obj, *sources, **kargs)`

This method is like *merge()* except that it accepts customizer which is invoked to produce the merged values of the destination and source properties. If customizer returns *None*, merging is handled by this method instead. The customizer is invoked with five arguments: (*obj_value*, *src_value*, *key*, *obj*, *source*).

Parameters

- **obj** (*dict*) – Destination object to merge source(s) into.
- **sources** (*dict*) – Source objects to merge from. subsequent sources overwrite previous ones.

Keyword Arguments *iteratee* (*function*, *optional*) – Iteratee function to handle merging (must be passed in as keyword argument).

Returns Merged object.

Return type dict

Warning: *obj* is modified in place.

Example

```
>>> cbk = lambda obj_val, src_val: obj_val + src_val
>>> obj1 = {'a': [1], 'b': [2]}
>>> obj2 = {'a': [3], 'b': [4]}
>>> res = merge_with(obj1, obj2, cbk)
>>> obj1 == {'a': [1, 3], 'b': [2, 4]}
True
```

New in version 4.0.0.

`pydash.objects.omit(obj, *properties)`

The opposite of *pick()*. This method creates an object composed of the property paths of *obj* that are not omitted.

Parameters

- **obj** (*mixed*) – Object to process.
- **properties** (*str*) – Property values to omit.

Returns Results of omitting properties.

Return type dict

Example

```
>>> omit({'a': 1, 'b': 2, 'c': 3}, 'b', 'c') == {'a': 1}
True
>>> omit({'a': 1, 'b': 2, 'c': 3}, ['a', 'c']) == {'b': 2}
True
>>> omit([1, 2, 3, 4], 0, 3) == {1: 2, 2: 3}
True
```

New in version 1.0.0.

Changed in version 4.0.0: Moved iteratee argument to `omit_by()`.

`pydash.objects.omit_by(obj, iteratee=None)`

The opposite of `pick_by()`. This method creates an object composed of the string keyed properties of object that predicate doesn't return truthy for. The predicate is invoked with two arguments: (value, key).

Parameters

- **obj** (*mixed*) – Object to process.
- **iteratee** (*mixed, optional*) – Iteratee used to determine which properties to omit.

Returns Results of omitting properties.

Return type dict

Example

```
>>> omit_by({'a': 1, 'b': '2', 'c': 3}, lambda v: isinstance(v, int))
{'b': '2'}
```

New in version 4.0.0.

`pydash.objects.parse_int(value, radix=None)`

Converts the given *value* into an integer of the specified *radix*. If *radix* is falsey, a radix of 10 is used unless the *value* is a hexadecimal, in which case a radix of 16 is used.

Parameters

- **value** (*mixed*) – Value to parse.
- **radix** (*int, optional*) – Base to convert to.

Returns Integer if parsable else None.

Return type mixed

Example

```
>>> parse_int('5')
5
>>> parse_int('12', 8)
10
>>> parse_int('x') is None
True
```

New in version 1.0.0.

`pydash.objects.pick(obj, *properties)`

Creates an object composed of the picked object properties.

Parameters

- **obj** (*list* | *dict*) – Object to pick from.
- **properties** (*str*) – Property values to pick.

Returns Dict containing picked properties.

Return type dict

Example

```
>>> pick({'a': 1, 'b': 2, 'c': 3}, 'a', 'b') == {'a': 1, 'b': 2}
True
```

New in version 1.0.0.

Changed in version 4.0.0: Moved iteratee argument to `pick_by()`.

`pydash.objects.pick_by(obj, iteratee=None)`

Creates an object composed of the object properties predicate returns truthy for. The predicate is invoked with two arguments: (value, key).

Parameters

- **obj** (*list* | *dict*) – Object to pick from.
- **iteratee** (*mixed*, *optional*) – Iteratee used to determine which properties to pick.

Returns Dict containing picked properties.

Return type dict

Example

```
>>> obj = {'a': 1, 'b': '2', 'c': 3 }
>>> pick_by(obj, lambda v: isinstance(v, int)) == {'a': 1, 'c': 3}
True
```

New in version 4.0.0.

`pydash.objects.rename_keys(obj, key_map)`

Rename the keys of *obj* using *key_map* and return new object.

Parameters

- **obj** (*dict*) – Object to rename.
- **key_map** (*dict*) – Renaming map whose keys correspond to existing keys in *obj* and whose values are the new key name.

Returns Renamed *obj*.

Return type dict

Example

```
>>> obj = rename_keys({'a': 1, 'b': 2, 'c': 3}, {'a': 'A', 'b': 'B'})
>>> obj == {'A': 1, 'B': 2, 'c': 3}
True
```

New in version 2.0.0.

`pydash.objects.set_(obj, path, value)`

Sets the value of an object described by *path*. If any part of the object path doesn't exist, it will be created.

Parameters

- **obj** (*list* / *dict*) – Object to modify.
- **path** (*str* | *list*) – Target path to set value to.
- **value** (*mixed*) – Value to set.

Returns Modified *obj*.

Return type mixed

Warning: *obj* is modified in place.

Example

```
>>> set_({}, 'a.b.c', 1)
{'a': {'b': {'c': 1}}}
>>> set_({}, 'a.0.c', 1)
{'a': {'0': {'c': 1}}}
>>> set_([1, 2], '[2][0]', 1)
[1, 2, [1]]
>>> set_({}, 'a.b[0].c', 1)
{'a': {'b': [{'c': 1}]}}
```

New in version 2.2.0.

Changed in version 3.3.0: Added `set_()` as main definition and `deep_set()` as alias.

Changed in version 4.0.0: Modify *obj* in place. Support creating default path values as *list* or *dict* based on whether key or index substrings are used. Remove alias `deep_set`.

`pydash.objects.set_with(obj, path, value, customizer=None)`

This method is like `set_()` except that it accepts *customizer* which is invoked to produce the objects of *path*. If *customizer* returns undefined path creation is handled by the method instead. The *customizer* is invoked with three arguments: (*nested_value*, *key*, *nested_object*).

Parameters

- **obj** (*list / dict*) – Object to modify.
- **path** (*str / list*) – Target path to set value to.
- **value** (*mixed*) – Value to set.
- **customizer** (*function, optional*) – The function to customize assigned values.

Returns Modified *obj*.

Return type mixed

Warning: *obj* is modified in place.

Example

```
>>> set_with({}, '[0][1]', 'a', lambda: {})
{0: {1: 'a'}}
```

New in version 4.0.0.

`pydash.objects.to_boolean(obj, true_values=('true', '1'), false_values=('false', '0'))`

Convert *obj* to boolean. This is not like the builtin `bool` function. By default commonly considered strings values are converted to their boolean equivalent, i.e., `'0'` and `'false'` are converted to `False` while `'1'` and `'true'` are converted to `True`. If a string value is provided that isn't recognized as having a common boolean conversion, then the returned value is `None`. Non-string values of *obj* are converted using `bool`. Optionally, *true_values* and *false_values* can be overridden but each value must be a string.

Parameters

- **obj** (*mixed*) – Object to convert.
- **true_values** (*tuple, optional*) – Values to consider `True`. Each value must be a string. Comparison is case-insensitive. Defaults to `('true', '1')`.
- **false_values** (*tuple, optional*) – Values to consider `False`. Each value must be a string. Comparison is case-insensitive. Defaults to `('false', '0')`.

Returns Boolean value of *obj*.

Return type bool

Example

```
>>> to_boolean('true')
True
>>> to_boolean('1')
True
>>> to_boolean('false')
False
>>> to_boolean('0')
False
>>> assert to_boolean('a') is None
```

New in version 3.0.0.

`pydash.objects.to_dict(obj)`

Convert *obj* to dict by created a new dict using *obj* keys and values.

Parameters *obj* – (mixed): Object to convert.

Returns Object converted to dict.

Return type dict

Example

```
>>> obj = {'a': 1, 'b': 2}
>>> obj2 = to_dict(obj)
>>> obj2 == obj
True
>>> obj2 is not obj
True
```

New in version 3.0.0.

Changed in version 4.0.0: Removed alias `to_plain_object`.

`pydash.objects.to_integer(obj)`

Converts *obj* to an integer.

Parameters *obj* (*str/int/float*) – Object to convert.

Returns Converted integer or 0 if it can't be converted.

Return type int

Example

```
>>> to_integer(3.2)
3
>>> to_integer('3.2')
3
>>> to_integer('3.9')
3
>>> to_integer('invalid')
0
```

New in version 4.0.0.

`pydash.objects.to_number(obj, precision=0)`

Convert *obj* to a number. All numbers are returned as `float`. If *precision* is negative, round *obj* to the nearest positive integer place. If *obj* can't be converted to a number, `None` is returned.

Parameters

- **obj** (*str/int/float*) – Object to convert.
- **precision** (*int*, *optional*) – Precision to round number to. Defaults to 0.

Returns Converted number or `None` if can't be converted.

Return type float

Example

```
>>> to_number('1234.5678')
1235.0
>>> to_number('1234.5678', 4)
1234.5678
>>> to_number(1, 2)
1.0
```

New in version 3.0.0.

`pydash.objects.to_pairs(obj)`

Creates a two dimensional list of an object's key-value pairs, i.e. `[[key1, value1], [key2, value2]]`.

Parameters `obj` (*mixed*) – Object to process.

Returns Two dimensional list of object's key-value pairs.

Return type list

Example

```
>>> to_pairs([1, 2, 3, 4])
[[0, 1], [1, 2], [2, 3], [3, 4]]
>>> to_pairs({'a': 1})
[['a', 1]]
```

New in version 1.0.0.

Changed in version 4.0.0: Renamed from `pairs` to `to_pairs`.

`pydash.objects.to_string(obj)`

Converts an object to string.

Parameters `obj` (*mixed*) – Object to convert.

Returns String representation of `obj`.

Return type str

Example

```
>>> to_string(1) == '1'
True
>>> to_string(None) == ''
True
>>> to_string([1, 2, 3]) == '[1, 2, 3]'
True
>>> to_string('a') == 'a'
True
```

New in version 2.0.0.

Changed in version 3.0.0: Convert `None` to empty string.

`pydash.objects.transform(obj, iteratee=None, accumulator=None)`

An alternative to `pydash.collections.reduce()`, this method transforms *obj* to a new accumulator object which is the result of running each of its properties through a *iteratee*, with each *iteratee* execution potentially mutating the accumulator object. The *iteratee* is invoked with four arguments: (*accumulator*, *value*, *key*, *object*). *Iteratees* may exit iteration early by explicitly returning `False`.

Parameters

- **obj** (*list/dict*) – Object to process.
- **iteratee** (*mixed*) – Iteratee applied per iteration.
- **accumulator** (*mixed, optional*) – Accumulated object. Defaults to `list`.

Returns Accumulated object.

Return type `mixed`

Example

```
>>> transform([1, 2, 3, 4], lambda acc, v, k: acc.append((k, v)))
[(0, 1), (1, 2), (2, 3), (3, 4)]
```

New in version 1.0.0.

`pydash.objects.unset(obj, path)`

Removes the property at *path* of *obj*.

Note: Only `list`, `dict`, or objects with a `pop()` method can be unset by this function.

Parameters

- **obj** (*mixed*) – The object to modify.
- **path** (*mixed*) – The path of the property to unset.

Returns Whether the property was deleted.

Return type `bool`

Warning: *obj* is modified in place.

Example

```
>>> obj = {'a': [{'b': {'c': 7}}]}
>>> unset(obj, 'a[0].b.c')
True
>>> obj
{'a': [{'b': {}}]}
>>> unset(obj, 'a[0].b.c')
False
```

`pydash.objects.update(obj, path, updater)`

This method is like `set_()` except that accepts updater to produce the value to set. Use `update_with()` to customize path creation. The updater is invoked with one argument: `(value)`.

Parameters

- **obj** (*list*/*dict*) – Object to modify.
- **path** (*str*/*list*) – A string or list of keys that describe the object path to modify.
- **updater** (*function*) – Function that returns updated value.

Returns Updated *obj*.

Return type mixed

Warning: *obj* is modified in place.

Example

```
>>> update({}, ['a', 'b'], lambda value: value)
{'a': {'b': None}}
>>> update([], [0, 0], lambda value: 1)
[[1]]
```

New in version 4.0.0.

`pydash.objects.update_with(obj, path, updater, customizer=None)`

This method is like `update()` except that it accepts customizer which is invoked to produce the objects of path. If customizer returns `None`, path creation is handled by the method instead. The customizer is invoked with three arguments: `(nested_value, key, nested_object)`.

Parameters

- **obj** (*list*/*dict*) – Object to modify.
- **path** (*str*/*list*) – A string or list of keys that describe the object path to modify.
- **updater** (*function*) – Function that returns updated value.
- **customizer** (*function*, *optional*) – The function to customize assigned values.

Returns Updated *obj*.

Return type mixed

Warning: *obj* is modified in place.

Example

```
>>> update_with({}, '[0][1]', lambda: 'a', lambda: {})
{0: {1: 'a'}}
```

New in version 4.0.0.

`pydash.objects.values(obj)`

Creates a list composed of the values of *obj*.

Parameters `obj` (*mixed*) – Object to extract values from.

Returns List of values.

Return type list

Example

```
>>> results = values({'a': 1, 'b': 2, 'c': 3})
>>> set(results) == set([1, 2, 3])
True
>>> values([2, 4, 6, 8])
[2, 4, 6, 8]
```

New in version 1.0.0.

Changed in version 1.1.0: Added `values_in` as alias.

Changed in version 4.0.0: Removed alias `values_in`.

Predicates

Predicate functions that return boolean evaluations of objects.

New in version 2.0.0.

`pydash.predicates.eq(value, other)`

Checks if `value` is equal to `other`.

Parameters

- **value** (*mixed*) – Value to compare.
- **other** (*mixed*) – Other value to compare.

Returns Whether `value` is equal to `other`.

Return type bool

Example

```
>>> eq(None, None)
True
>>> eq(None, '')
False
>>> eq('a', 'a')
True
>>> eq(1, str(1))
False
```

New in version 4.0.0.

`pydash.predicates.gt(value, other)`

Checks if `value` is greater than `other`.

Parameters

- **value** (*number*) – Value to compare.

- **other** (*number*) – Other value to compare.

Returns Whether *value* is greater than *other*.

Return type bool

Example

```
>>> gt(5, 3)
True
>>> gt(3, 5)
False
>>> gt(5, 5)
False
```

New in version 3.3.0.

`pydash.predicates.gte` (*value*, *other*)

Checks if *value* is greater than or equal to *other*.

Parameters

- **value** (*number*) – Value to compare.
- **other** (*number*) – Other value to compare.

Returns Whether *value* is greater than or equal to *other*.

Return type bool

Example

```
>>> gte(5, 3)
True
>>> gte(3, 5)
False
>>> gte(5, 5)
True
```

New in version 3.3.0.

`pydash.predicates.lt` (*value*, *other*)

Checks if *value* is less than *other*.

Parameters

- **value** (*number*) – Value to compare.
- **other** (*number*) – Other value to compare.

Returns Whether *value* is less than *other*.

Return type bool

Example

```
>>> lt(5, 3)
False
>>> lt(3, 5)
True
>>> lt(5, 5)
False
```

New in version 3.3.0.

`pydash.predicates.lte(value, other)`

Checks if *value* is less than or equal to *other*.

Parameters

- **value** (*number*) – Value to compare.
- **other** (*number*) – Other value to compare.

Returns Whether *value* is less than or equal to *other*.

Return type bool

Example

```
>>> lte(5, 3)
False
>>> lte(3, 5)
True
>>> lte(5, 5)
True
```

New in version 3.3.0.

`pydash.predicates.in_range(value, start=0, end=None)`

Checks if *value* is between *start* and up to but not including *end*. If *end* is not specified it defaults to *start* with *start* becoming 0.

Parameters

- **value** (*int/float*) – Number to check.
- **start** (*int/float, optional*) – Start of range inclusive. Defaults to 0.
- **end** (*int/float, optional*) – End of range exclusive. Defaults to *start*.

Returns Whether *value* is in range.

Return type bool

Example

```
>>> in_range(2, 4)
True
>>> in_range(4, 2)
False
>>> in_range(2, 1, 3)
True
>>> in_range(3, 1, 2)
False
```



```
>>> in_range(2.5, 3.5)
True
>>> in_range(3.5, 2.5)
False
```

New in version 3.1.0.

`pydash.predicates.is_associative(value)`

Checks if *value* is an associative object meaning that it can be accessed via an index or key

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is associative.

Return type bool

Example

```
>>> is_associative([])
True
>>> is_associative({})
True
>>> is_associative(1)
False
>>> is_associative(True)
False
```

New in version 2.0.0.

`pydash.predicates.is_blank(text)`

Checks if *text* contains only whitespace characters.

Parameters *text* (*str*) – String to test.

Returns Whether *text* is blank.

Return type bool

Example

```
>>> is_blank(' ')
True
>>> is_blank(' \r\n ')
True
>>> is_blank(False)
False
```

New in version 3.0.0.

`pydash.predicates.is_boolean(value)`

Checks if *value* is a boolean value.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is a boolean.

Return type bool

Example

```
>>> is_boolean(True)
True
>>> is_boolean(False)
True
>>> is_boolean(0)
False
```

New in version 1.0.0.

Changed in version 3.0.0: Added `is_bool` as alias.

Changed in version 4.0.0: Removed alias `is_bool`.

`pydash.predicates.is_builtin(value)`

Checks if *value* is a Python builtin function or method.

Parameters *value* (*function*) – Value to check.

Returns Whether *value* is a Python builtin function or method.

Return type bool

Example

```
>>> is_builtin(1)
True
>>> is_builtin(list)
True
>>> is_builtin('foo')
False
```

New in version 3.0.0.

Changed in version 4.0.0: Removed alias `is_native`.

`pydash.predicates.is_date(value)`

Check if *value* is a date object.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is a date object.

Return type bool

Example

```
>>> import datetime
>>> is_date(datetime.date.today())
True
>>> is_date(datetime.datetime.today())
True
>>> is_date('2014-01-01')
False
```

Note: This will also return `True` for datetime objects.

New in version 1.0.0.

`pydash.predicates.is_decreasing(value)`

Check if *value* is monotonically decreasing.

Parameters *value* (*list*) – Value to check.

Returns Whether *value* is monotonically decreasing.

Return type bool

Example

```
>>> is_decreasing([5, 4, 4, 3])
True
>>> is_decreasing([5, 5, 5])
True
>>> is_decreasing([5, 4, 5])
False
```

New in version 2.0.0.

`pydash.predicates.is_dict(value)`

Checks if *value* is a dict.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is a dict.

Return type bool

Example

```
>>> is_dict({})
True
>>> is_dict([])
False
```

New in version 1.0.0.

Changed in version 3.0.0: Added `is_dict()` as main definition and made `is_plain_object` an alias.

Changed in version 4.0.0: Removed alias `is_plain_object`.

`pydash.predicates.is_empty(value)`

Checks if *value* is empty.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is empty.

Return type bool

Example

```
>>> is_empty(0)
True
>>> is_empty(1)
True
>>> is_empty(True)
True
>>> is_empty('foo')
False
>>> is_empty(None)
True
>>> is_empty({})
True
```

Note: Returns True for booleans and numbers.

New in version 1.0.0.

`pydash.predicates.is_equal(value, other)`

Performs a comparison between two values to determine if they are equivalent to each other.

Parameters

- **value** (*list/dict*) – Object to compare.
- **other** (*list/dict*) – Object to compare.

Returns Whether *value* and *other* are equal.

Return type bool

Example

```
>>> is_equal([1, 2, 3], [1, 2, 3])
True
>>> is_equal('a', 'A')
False
```

New in version 1.0.0.

Changed in version 4.0.0: Removed *iteratee* from `is_equal()` and added it in `is_equal_with()`.

`pydash.predicates.is_equal_with(value, other, customizer)`

This method is like `_.isEqual` except that it accepts *customizer* which is invoked to compare values. A *customizer* is provided which will be executed to compare values. If the *customizer* returns `None`, comparisons will be handled by the method instead. The *customizer* is invoked with two arguments: (*value*, *other*).

Parameters

- **value** (*list/dict*) – Object to compare.
- **other** (*list/dict*) – Object to compare.
- **customizer** (*mixed, optional*) – Customizer used to compare values from *value* and *other*.

Returns Whether *value* and *other* are equal.

Return type bool

Example

```
>>> is_equal_with([1, 2, 3], [1, 2, 3], None)
True
>>> is_equal_with('a', 'A', None)
False
>>> is_equal_with('a', 'A', lambda a, b: a.lower() == b.lower())
True
```

New in version 4.0.0.

`pydash.predicates.is_error(value)`

Checks if *value* is an Exception.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is an exception.

Return type bool

Example

```
>>> is_error(Exception())
True
>>> is_error(Exception)
False
>>> is_error(None)
False
```

New in version 1.1.0.

`pydash.predicates.is_even(value)`

Checks if *value* is even.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is even.

Return type bool

Example

```
>>> is_even(2)
True
>>> is_even(3)
False
>>> is_even(False)
False
```

New in version 2.0.0.

`pydash.predicates.is_float(value)`

Checks if *value* is a float.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is a float.

Return type bool

Example

```
>>> is_float(1.0)
True
>>> is_float(1)
False
```

New in version 2.0.0.

`pydash.predicates.is_function(value)`

Checks if *value* is a function.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is callable.

Return type bool

Example

```
>>> is_function(list)
True
>>> is_function(lambda: True)
True
>>> is_function(1)
False
```

New in version 1.0.0.

`pydash.predicates.is_increasing(value)`

Check if *value* is monotonically increasing.

Parameters *value* (*list*) – Value to check.

Returns Whether *value* is monotonically increasing.

Return type bool

Example

```
>>> is_increasing([1, 3, 5])
True
>>> is_increasing([1, 1, 2, 3, 3])
True
>>> is_increasing([5, 5, 5])
True
>>> is_increasing([1, 2, 4, 3])
False
```

New in version 2.0.0.

`pydash.predicates.is_indexed(value)`

Checks if *value* is integer indexed, i.e., list, str or tuple.

Parameters **value** (*mixed*) – Value to check.

Returns Whether *value* is integer indexed.

Return type bool

Example

```
>>> is_indexed('')
True
>>> is_indexed([])
True
>>> is_indexed(())
True
>>> is_indexed({})
False
```

New in version 2.0.0.

Changed in version 3.0.0: Return True for tuples.

`pydash.predicates.is_instance_of(value, types)`

Checks if *value* is an instance of *types*.

Parameters

- **value** (*mixed*) – Value to check.
- **types** (*mixed*) – Types to check against. Pass as tuple to check if *value* is one of multiple types.

Returns Whether *value* is an instance of *types*.

Return type bool

Example

```
>>> is_instance_of({}, dict)
True
>>> is_instance_of({}, list)
False
```

New in version 2.0.0.

`pydash.predicates.is_integer(value)`

Checks if *value* is a integer.

Parameters **value** (*mixed*) – Value to check.

Returns Whether *value* is an integer.

Return type bool

Example

```
>>> is_integer(1)
True
>>> is_integer(1.0)
False
>>> is_integer(True)
False
```

New in version 2.0.0.

Changed in version 3.0.0: Added `is_int` as alias.

Changed in version 4.0.0: Removed alias `is_int`.

`pydash.predicates.is_iterable(value)`

Checks if *value* is an iterable.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is an iterable.

Return type bool

Example

```
>>> is_iterable([])
True
>>> is_iterable({})
True
>>> is_iterable(())
True
>>> is_iterable(5)
False
>>> is_iterable(True)
False
```

New in version 3.3.0.

`pydash.predicates.is_json(value)`

Checks if *value* is a valid JSON string.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is JSON.

Return type bool

Example

```
>>> is_json({})
False
>>> is_json('{}')
True
>>> is_json({"hello": 1, "world": 2})
False
>>> is_json('{"hello": 1, "world": 2}')
```

New in version 2.0.0.

`pydash.predicates.is_list(value)`

Checks if *value* is a list.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is a list.

Return type bool

Example

```
>>> is_list([])
True
>>> is_list({})
False
>>> is_list(())
False
```

New in version 1.0.0.

`pydash.predicates.is_match(obj, source)`

Performs a partial deep comparison between *obj* and *source* to determine if *obj* contains equivalent property values.

Parameters

- **obj** (*list|dict*) – Object to compare.
- **source** (*list|dict*) – Object of property values to match.

Returns Whether *obj* is a match or not.

Return type bool

Example

```
>>> is_match({'a': 1, 'b': 2}, {'b': 2})
True
>>> is_match({'a': 1, 'b': 2}, {'b': 3})
False
>>> is_match({'a': [{'b': [{'c': 3, 'd': 4}]}]}, {'a': [{'b': 4}]})
True
```

New in version 3.0.0.

Changed in version 3.2.0: Don't compare *obj* and *source* using type. Use `isinstance` exclusively.

Changed in version 4.0.0: Move *iteratee* argument to `is_match_with()`.

`pydash.predicates.is_match_with(obj, source, customizer=None, _key=<pydash.helpers._NoValue object>, _obj=<pydash.helpers._NoValue object>, _source=<pydash.helpers._NoValue object>)`

This method is like `is_match()` except that it accepts *customizer* which is invoked to compare values. If *customizer* returns `None`, comparisons are handled by the method instead. The *customizer* is invoked with five arguments: (*obj_value*, *src_value*, *index|key*, *obj*, *source*).

Parameters

- **obj** (*list* / *dict*) – Object to compare.
- **source** (*list* / *dict*) – Object of property values to match.
- **customizer** (*mixed*, *optional*) – Customizer used to compare values from *obj* and *source*.

Returns Whether *obj* is a match or not.

Return type bool

Example

```
>>> is_greeting = lambda val: val in ('hello', 'hi')
>>> customizer = lambda ov, sv: is_greeting(ov) and is_greeting(sv)
>>> obj = {'greeting': 'hello'}
>>> src = {'greeting': 'hi'}
>>> is_match_with(obj, src, customizer)
True
```

New in version 4.0.0.

`pydash.predicates.is_monotone` (*value*, *op*)

Checks if *value* is monotonic when *operator* used for comparison.

Parameters

- **value** (*list*) – Value to check.
- **op** (*function*) – Operation to used for comparison.

Returns Whether *value* is monotone.

Return type bool

Example

```
>>> is_monotone([1, 1, 2, 3], operator.le)
True
>>> is_monotone([1, 1, 2, 3], operator.lt)
False
```

New in version 2.0.0.

`pydash.predicates.is_nan` (*value*)

Checks if *value* is not a number.

Parameters **value** (*mixed*) – Value to check.

Returns Whether *value* is not a number.

Return type bool

Example

```
>>> is_nan('a')
True
>>> is_nan(1)
False
>>> is_nan(1.0)
False
```

New in version 1.0.0.

`pydash.predicates.is_negative(value)`

Checks if *value* is negative.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is negative.

Return type bool

Example

```
>>> is_negative(-1)
True
>>> is_negative(0)
False
>>> is_negative(1)
False
```

New in version 2.0.0.

`pydash.predicates.is_none(value)`

Checks if *value* is *None*.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is *None*.

Return type bool

Example

```
>>> is_none(None)
True
>>> is_none(False)
False
```

New in version 1.0.0.

`pydash.predicates.is_number(value)`

Checks if *value* is a number.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is a number.

Return type bool

Note: Returns True for int, long (PY2), float, and decimal.Decimal.

Example

```
>>> is_number(1)
True
>>> is_number(1.0)
True
>>> is_number('a')
False
```

New in version 1.0.0.

Changed in version 3.0.0: Added `is_num` as alias.

Changed in version 4.0.0: Removed alias `is_num`.

`pydash.predicates.is_object(value)`

Checks if *value* is a list or dict.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is list or dict.

Return type bool

Example

```
>>> is_object([])
True
>>> is_object({})
True
>>> is_object(())
False
>>> is_object(1)
False
```

New in version 1.0.0.

`pydash.predicates.is_odd(value)`

Checks if *value* is odd.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is odd.

Return type bool

Example

```
>>> is_odd(3)
True
>>> is_odd(2)
False
>>> is_odd('a')
False
```

New in version 2.0.0.

`pydash.predicates.is_positive(value)`

Checks if *value* is positive.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is positive.

Return type bool

Example

```
>>> is_positive(1)
True
>>> is_positive(0)
False
>>> is_positive(-1)
False
```

New in version 2.0.0.

`pydash.predicates.is_reg_exp(value)`

Checks if *value* is a RegExp object.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is a RegExp object.

Return type bool

Example

```
>>> is_reg_exp(re.compile(''))
True
>>> is_reg_exp('')
False
```

New in version 1.1.0.

Changed in version 4.0.0: Removed alias `is_re`.

`pydash.predicates.is_set(value)`

Checks if the given value is a set object or not.

Parameters *value* (*mixed*) – Value passed in by the user.

Returns True if the given value is a set else False.

Return type bool

Example

```
>>> is_set(set([1, 2]))
True
>>> is_set([1, 2, 3])
False
```

New in version 4.0.0.

`pydash.predicates.is_strictly_decreasing(value)`

Check if *value* is strictly decreasing.

Parameters *value* (*list*) – Value to check.

Returns Whether *value* is strictly decreasing.

Return type bool

Example

```
>>> is_strictly_decreasing([4, 3, 2, 1])
True
>>> is_strictly_decreasing([4, 4, 2, 1])
False
```

New in version 2.0.0.

`pydash.predicates.is_strictly_increasing(value)`

Check if *value* is strictly increasing.

Parameters *value* (*list*) – Value to check.

Returns Whether *value* is strictly increasing.

Return type bool

Example

```
>>> is_strictly_increasing([1, 2, 3, 4])
True
>>> is_strictly_increasing([1, 1, 3, 4])
False
```

New in version 2.0.0.

`pydash.predicates.is_string(value)`

Checks if *value* is a string.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is a string.

Return type bool

Example

```
>>> is_string('')
True
>>> is_string(1)
False
```

New in version 1.0.0.

`pydash.predicates.is_tuple(value)`

Checks if *value* is a tuple.

Parameters *value* (*mixed*) – Value to check.

Returns Whether *value* is a tuple.

Return type bool

Example

```
>>> is_tuple(())
True
>>> is_tuple({})
False
>>> is_tuple([])
False
```

New in version 3.0.0.

`pydash.predicates.is_zero(value)`

Checks if *value* is 0.

Parameters **value** (*mixed*) – Value to check.

Returns Whether *value* is 0.

Return type bool

Example

```
>>> is_zero(0)
True
>>> is_zero(1)
False
```

New in version 2.0.0.

Strings

String functions.

New in version 1.1.0.

`pydash.strings.camel_case(text)`

Converts *text* to camel case.

Parameters **text** (*str*) – String to convert.

Returns String converted to camel case.

Return type str

Example

```
>>> camel_case('FOO BAR_bAz')
'fooBarBAz'
```

New in version 1.1.0.

`pydash.strings.capitalize(text, strict=True)`

Capitalizes the first character of *text*.

Parameters

- **text** (*str*) – String to capitalize.
- **strict** (*bool*, *optional*) – Whether to cast rest of string to lower case. Defaults to `True`.

Returns Capitalized string.

Return type `str`

Example

```
>>> capitalize('once upon a TIME')
'Once upon a time'
>>> capitalize('once upon a TIME', False)
'Once upon a TIME'
```

New in version 1.1.0.

Changed in version 3.0.0: Added *strict* option.

`pydash.strings.chop(text, step)`

Break up *text* into intervals of length *step*.

Parameters

- **text** (*str*) – String to chop.
- **step** (*int*) – Interval to chop *text*.

Returns

List of chopped characters. If *text* is *None* an empty list is returned.

Return type `list`

Example

```
>>> chop('abcdefg', 3)
['abc', 'def', 'g']
```

New in version 3.0.0.

`pydash.strings.chop_right(text, step)`

Like `chop()` except *text* is chopped from right.

Parameters

- **text** (*str*) – String to chop.
- **step** (*int*) – Interval to chop *text*.

Returns List of chopped characters.

Return type `list`

Example

```
>>> chop_right('abcdefg', 3)
['a', 'bcd', 'efg']
```

New in version 3.0.0.

`pydash.strings.chars(text)`

Split *text* into a list of single characters.

Parameters *text* (*str*) – String to split up.

Returns List of individual characters.

Return type list

Example

```
>>> chars('onetwo')
['o', 'n', 'e', 't', 'w', 'o']
```

New in version 3.0.0.

`pydash.strings.clean(text)`

Trim and replace multiple spaces with a single space.

Parameters *text* (*str*) – String to clean.

Returns Cleaned string.

Return type str

Example

```
>>> clean('a b c d')
'a b c d'
```

New in version 3.0.0.

`pydash.strings.count_substr(text, subtext)`

Count the occurrences of *subtext* in *text*.

Parameters

- **text** (*str*) – Source string to count from.
- **subtext** (*str*) – String to count.

Returns Number of occurrences of *subtext* in *text*.

Return type int

Example

```
>>> count_substr('aabbccddaabbccdd', 'bc')
2
```

New in version 3.0.0.

`pydash.strings.deburrr(text)`

Debursrs *text* by converting latin-1 supplementary letters to basic latin letters.

Parameters *text* (*str*) – String to deburr.

Returns Deburred string.

Return type *str*

Example

```
>>> deburr('déjà vu')
'...'
>>> 'déjà vu'
'deja vu'
```

New in version 2.0.0.

`pydash.strings.decapitalize(text)`

Decapitalizes the first character of *text*.

Parameters *text* (*str*) – String to decapitalize.

Returns Decapitalized string.

Return type *str*

Example

```
>>> decapitalize('FOO BAR')
'foo BAR'
```

New in version 3.0.0.

`pydash.strings.ends_with(text, target, position=None)`

Checks if *text* ends with a given target string.

Parameters

- **text** (*str*) – String to check.
- **target** (*str*) – String to check for.
- **position** (*int*, *optional*) – Position to search from. Defaults to end of *text*.

Returns Whether *text* ends with *target*.

Return type *bool*

Example

```
>>> ends_with('abc def', 'def')
True
>>> ends_with('abc def', 4)
False
```

New in version 1.1.0.

`pydash.strings.ensure_ends_with(text, suffix)`

Append a given suffix to a string, but only if the source string does not end with that suffix.

Parameters

- **text** (*str*) – Source string to append *suffix* to.
- **suffix** (*str*) – String to append to the source string if the source string does not end with *suffix*.

Returns source string possibly extended by *suffix*.

Return type `str`

Example

```
>>> ensure_ends_with('foo bar', '!')
'foo bar!'
>>> ensure_ends_with('foo bar!', '!')
'foo bar!'
```

New in version 2.4.0.

`pydash.strings.ensure_starts_with(text, prefix)`

Prepend a given prefix to a string, but only if the source string does not start with that prefix.

Parameters

- **text** (*str*) – Source string to prepend *prefix* to.
- **suffix** (*str*) – String to prepend to the source string if the source string does not start with *prefix*.

Returns source string possibly prefixed by *prefix*

Return type `str`

Example

```
>>> ensure_starts_with('foo bar', 'Oh my! ')
'Oh my! foo bar'
>>> ensure_starts_with('Oh my! foo bar', 'Oh my! ')
'Oh my! foo bar'
```

New in version 2.4.0.

`pydash.strings.escape(text)`

Converts the characters `&`, `<`, `>`, `"`, `'`, and `\`` in *text* to their corresponding HTML entities.

Parameters **text** (*str*) – String to escape.

Returns HTML escaped string.

Return type `str`

Example

```
>>> escape('"1 > 2 && 3 < 4"')
'&quot;1 &gt; 2 &amp;&amp; 3 &lt; 4&quot;'
```

New in version 1.0.0.

Changed in version 1.1.0: Moved function to `pydash.strings`.

`pydash.strings.escape_reg_exp(text)`

Escapes the RegExp special characters in *text*.

Parameters *text* (*str*) – String to escape.

Returns RegExp escaped string.

Return type *str*

Example

```
>>> escape_reg_exp('[() ]')
'\\[\\(\\)\\ ]'
```

New in version 1.1.0.

Changed in version 4.0.0: Removed alias `escape_re`

`pydash.strings.has_substr(text, subtext)`

Returns whether *subtext* is included in *text*.

Parameters

- **text** (*str*) – String to search.
- **subtext** (*str*) – String to search for.

Returns Whether *subtext* is found in *text*.

Return type *bool*

Example

```
>>> has_substr('abcdef', 'bc')
True
>>> has_substr('abcdef', 'bb')
False
```

New in version 3.0.0.

`pydash.strings.human_case(text)`

Converts *text* to human case which has only the first letter capitalized and each word separated by a space.

Parameters *text* (*str*) – String to convert.

Returns String converted to human case.

Return type *str*

Example

```
>>> human_case('abc-def_hij_lmn')
'Abc def hij_lmn'
>>> human_case('user_id')
'User'
```

New in version 3.0.0.

`pydash.strings.insert_substr(text, index, subtext)`
 Insert *subtext* in *text* starting at position *index*.

Parameters

- **text** (*str*) – String to add substring to.
- **index** (*int*) – String index to insert into.
- **subtext** (*str*) – String to insert.

Returns Modified string.

Return type str

Example

```
>>> insert_substr('abcdef', 3, '--')
'abc--def'
```

New in version 3.0.0.

`pydash.strings.join(array, separator='')`
 Joins an iterable into a string using *separator* between each element.

Parameters

- **array** (*iterable*) – Iterable to implode.
- **separator** (*str*, *optional*) – Separator to using when joining. Defaults to ' '.

Returns Joined string.

Return type str

Example

```
>>> join(['a', 'b', 'c']) == 'abc'
True
>>> join([1, 2, 3, 4], '&') == '1&2&3&4'
True
>>> join('abcdef', '-') == 'a-b-c-d-e-f'
True
```

New in version 2.0.0.

Changed in version 4.0.0: Removed alias `implode`.

`pydash.strings.kebab_case(text)`
 Converts *text* to kebab case (a.k.a. spinal case).

Parameters `text` (*str*) – String to convert.

Returns String converted to kebab case.

Return type `str`

Example

```
>>> kebab_case('a b c_d-e!f')
'a-b-c-d-e-f'
```

New in version 1.1.0.

`pydash.strings.lines` (*text*)

Split lines in *text* into an array.

Parameters `text` (*str*) – String to split.

Returns String split by lines.

Return type `list`

Example

```
>>> lines('a\nb\r\nc')
['a', 'b', 'c']
```

New in version 3.0.0.

`pydash.strings.lower_case` (*text*)

Converts string to lower case as space separated words.

Parameters `text` (*str*) – String to convert.

Returns String converted to lower case as space separated words.

Return type `str`

Example

```
>>> lower_case('fooBar')
'foo bar'
>>> lower_case('--foo-Bar--')
'foo bar'
>>> lower_case('/?*Foo10/;"B*Ar')
'foo 10 b ar'
```

New in version 4.0.0.

`pydash.strings.lower_first` (*text*)

Converts the first character of string to lower case.

Parameters `text` (*str*) – String passed in by the user.

Returns String in which the first character is converted to lower case.

Return type `str`

Example

```
>>> lower_first('FRED')
'fRED'
>>> lower_first('Foo Bar')
'foo Bar'
>>> lower_first('lfoobar')
'lfoobar'
>>> lower_first(';foobar')
';foobar'
```

New in version 4.0.0.

`pydash.strings.number_format` (*number*, *scale*=0, *decimal_separator*='.', *order_separator*=' ',)

Format a number to scale with custom decimal and order separators.

Parameters

- **number** (*int* / *float*) – Number to format.
- **scale** (*int*, *optional*) – Number of decimals to include. Defaults to 0.
- **decimal_separator** (*str*, *optional*) – Decimal separator to use. Defaults to '.'.
- **order_separator** (*str*, *optional*) – Order separator to use. Defaults to ' '.

Returns Formatted number as string.

Return type str

Example

```
>>> number_format(1234.5678)
'1,235'
>>> number_format(1234.5678, 2, ',', '.')
'1.234,57'
```

New in version 3.0.0.

`pydash.strings.pad` (*text*, *length*, *chars*=' ')

Pads *text* on the left and right sides if it is shorter than the given padding length. The *chars* string may be truncated if the number of padding characters can't be evenly divided by the padding length.

Parameters

- **text** (*str*) – String to pad.
- **length** (*int*) – Amount to pad.
- **chars** (*str*, *optional*) – Characters to pad with. Defaults to " ".

Returns Padded string.

Return type str

Example

```
>>> pad('abc', 5)
' abc '
>>> pad('abc', 6, 'x')
'xabcxx'
>>> pad('abc', 5, '...')
'.abc.'
```

New in version 1.1.0.

Changed in version 3.0.0: Fix handling of multiple *chars* so that padded string isn't over padded.

`pydash.strings.pad_end(text, length, chars='')`

Pads *text* on the right side if it is shorter than the given padding length. The *chars* string may be truncated if the number of padding characters can't be evenly divided by the padding length.

Parameters

- **text** (*str*) – String to pad.
- **length** (*int*) – Amount to pad.
- **chars** (*str*, *optional*) – Characters to pad with. Defaults to " ".

Returns Padded string.

Return type `str`

Example

```
>>> pad_end('abc', 5)
'abc '
>>> pad_end('abc', 5, '..')
'abc..'
```

New in version 1.1.0.

Changed in version 4.0.0: Renamed from `pad_right` to `pad_end`.

`pydash.strings.pad_start(text, length, chars='')`

Pads *text* on the left side if it is shorter than the given padding length. The *chars* string may be truncated if the number of padding characters can't be evenly divided by the padding length.

Parameters

- **text** (*str*) – String to pad.
- **length** (*int*) – Amount to pad.
- **chars** (*str*, *optional*) – Characters to pad with. Defaults to " ".

Returns Padded string.

Return type `str`

Example


```
>>> pad_start('abc', 5)
'   abc'
>>> pad_start('abc', 5, '.')
'..abc'
```

New in version 1.1.0.

Changed in version 4.0.0: Renamed from `pad_left` to `pad_start`.

`pydash.strings.pascal_case(text, strict=True)`

Like `camel_case()` except the first letter is capitalized.

Parameters

- **text** (*str*) – String to convert.
- **strict** (*bool*, *optional*) – Whether to cast rest of string to lower case. Defaults to `True`.

Returns String converted to class case.

Return type `str`

Example

```
>>> pascal_case('FOO BAR_bAz')
'FooBarBaz'
>>> pascal_case('FOO BAR_bAz', False)
'FooBarBAz'
```

New in version 3.0.0.

`pydash.strings.predecessor(char)`

Return the predecessor character of *char*.

Parameters **char** (*str*) – Character to find the predecessor of.

Returns Predecessor character.

Return type `str`

Example

```
>>> predecessor('c')
'b'
>>> predecessor('C')
'B'
>>> predecessor('3')
'2'
```

New in version 3.0.0.

`pydash.strings.prune(text, length=0, omission='...')`

Like `truncate()` except it ensures that the pruned string doesn't exceed the original length, i.e., it avoids half-chopped words when truncating. If the pruned text + *omission* text is longer than the original text, then the original text is returned.

Parameters

- **text** (*str*) – String to prune.
- **length** (*int*, *optional*) – Target prune length. Defaults to 0.
- **omission** (*str*, *optional*) – Omission text to append to the end of the pruned string. Defaults to '... '.

Returns Pruned string.

Return type str

Example

```
>>> prune('Fe fi fo fum', 5)
'Fe fi...'
>>> prune('Fe fi fo fum', 6)
'Fe fi...'
>>> prune('Fe fi fo fum', 7)
'Fe fi...'
>>> prune('Fe fi fo fum', 8, ',,,')
'Fe fi fo,,,'
```

New in version 3.0.0.

`pydash.strings.quote(text, quote_char=“”)`

Quote a string with another string.

Parameters

- **text** (*str*) – String to be quoted.
- **quote_char** (*str*, *optional*) – the quote character. Defaults to “”.

Returns the quoted string.

Return type str

Example

```
>>> quote('To be or not to be')
'"To be or not to be"'
>>> quote('To be or not to be', "'")
"'To be or not to be'"
```

New in version 2.4.0.

`pydash.strings.reg_exp_js_match(text, reg_exp)`

Return list of matches using Javascript style regular expression.

Parameters

- **text** (*str*) – String to evaluate.
- **reg_exp** (*str*) – Javascript style regular expression.

Returns List of matches.

Return type list

Example

```
>>> reg_exp_js_match('aaBBcc', '/bb/')
[]
>>> reg_exp_js_match('aaBBcc', '/bb/i')
['BB']
>>> reg_exp_js_match('aaBBccbb', '/bb/i')
['BB']
>>> reg_exp_js_match('aaBBccbb', '/bb/gi')
['BB', 'bb']
```

New in version 2.0.0.

Changed in version 3.0.0: Reordered arguments to make *text* first.

Changed in version 4.0.0: Renamed from `js_match` to `reg_exp_js_match`.

`pydash.strings.reg_exp_js_replace(text, reg_exp, repl)`

Replace *text* with *repl* using Javascript style regular expression to find matches.

Parameters

- **text** (*str*) – String to evaluate.
- **reg_exp** (*str*) – Javascript style regular expression.
- **repl** (*str*) – Replacement string.

Returns Modified string.

Return type `str`

Example

```
>>> reg_exp_js_replace('aaBBcc', '/bb/', 'X')
'aaBBcc'
>>> reg_exp_js_replace('aaBBcc', '/bb/i', 'X')
'aaXcc'
>>> reg_exp_js_replace('aaBBccbb', '/bb/i', 'X')
'aaXccbb'
>>> reg_exp_js_replace('aaBBccbb', '/bb/gi', 'X')
'aaXccX'
```

New in version 2.0.0.

Changed in version 3.0.0: Reordered arguments to make *text* first.

Changed in version 4.0.0: Renamed from `js_replace` to `reg_exp_js_replace`.

`pydash.strings.reg_exp_replace(text, pattern, repl, ignore_case=False, count=0)`

Replace occurrences of regex *pattern* with *repl* in *text*. Optionally, ignore case when replacing. Optionally, set *count* to limit number of replacements.

Parameters

- **text** (*str*) – String to replace.
- **pattern** (*str*) – String pattern to find and replace.
- **repl** (*str*) – String to substitute *pattern* with.

- **ignore_case** (*bool, optional*) – Whether to ignore case when replacing. Defaults to `False`.
- **count** (*int, optional*) – Maximum number of occurrences to replace. Defaults to 0 which replaces all.

Returns Replaced string.

Return type `str`

Example

```
>>> reg_exp_replace('aabbcc', 'b', 'X')
'aaXXcc'
>>> reg_exp_replace('aabbcc', 'B', 'X', ignore_case=True)
'aaXXcc'
>>> reg_exp_replace('aabbcc', 'b', 'X', count=1)
'aaXbcc'
>>> reg_exp_replace('aabbcc', '[ab]', 'X')
'XXXXcc'
```

New in version 3.0.0.

Changed in version 4.0.0: Renamed from `re_replace` to `reg_exp_replace`.

`pydash.strings.repeat` (*text, n=0*)

Repeats the given string *n* times.

Parameters

- **text** (*str*) – String to repeat.
- **n** (*int, optional*) – Number of times to repeat the string.

Returns Repeated string.

Return type `str`

Example

```
>>> repeat('.', 5)
'.....'
```

New in version 1.1.0.

`pydash.strings.replace` (*text, pattern, repl, ignore_case=False, count=0, escape=True*)

Replace occurrences of *pattern* with *repl* in *text*. Optionally, ignore case when replacing. Optionally, set *count* to limit number of replacements.

Parameters

- **text** (*str*) – String to replace.
- **pattern** (*str*) – String pattern to find and replace.
- **repl** (*str*) – String to substitute *pattern* with.
- **ignore_case** (*bool, optional*) – Whether to ignore case when replacing. Defaults to `False`.

- **count** (*int*, *optional*) – Maximum number of occurrences to replace. Defaults to 0 which replaces all.
- **escape** (*bool*, *optional*) – Whether to escape *pattern* when searching. This is needed if a literal replacement is desired when *pattern* may contain special regular expression characters. Defaults to `True`.

Returns Replaced string.

Return type `str`

Example

```
>>> replace('aabbcc', 'b', 'X')
'aaXXcc'
>>> replace('aabbcc', 'B', 'X', ignore_case=True)
'aaXXcc'
>>> replace('aabbcc', 'b', 'X', count=1)
'aaXbcc'
>>> replace('aabbcc', '[ab]', 'X')
'aabbcc'
>>> replace('aabbcc', '[ab]', 'X', escape=False)
'XXXXcc'
```

New in version 3.0.0.

`pydash.strings.separator_case(text, separator)`

Splits *text* on words and joins with *separator*.

Parameters

- **text** (*str*) – String to convert.
- **separator** (*str*) – Separator to join words with.

Returns Converted string.

Return type `str`

Example

```
>>> separator_case('a!!b___c.d', '-')
'a-b-c-d'
```

New in version 3.0.0.

`pydash.strings.series_phrase(items, separator=',', last_separator=' and ', serial=False)`

Join items into a grammatical series phrase, e.g., "item1, item2, item3 and item4".

Parameters

- **items** (*list*) – List of string items to join.
- **separator** (*str*, *optional*) – Item separator. Defaults to `,`.
- **last_separator** (*str*, *optional*) – Last item separator. Defaults to `' and '`.
- **serial** (*bool*, *optional*) – Whether to include *separator* with *last_separator* when number of items is greater than 2. Defaults to `False`.

Returns Joined string.

Return type str

Example

Example:

```
>>> series_phrase(['apples', 'bananas', 'peaches'])
'apples, bananas and peaches'
>>> series_phrase(['apples', 'bananas', 'peaches'], serial=True)
'apples, bananas, and peaches'
>>> series_phrase(['apples', 'bananas', 'peaches'], '; ', ', or ')
'apples; bananas, or peaches'
```

New in version 3.0.0.

`pydash.strings.series_phrase_serial(items, separator=',', last_separator=' and ')`

Join items into a grammatical series phrase using a serial separator, e.g., "item1, item2, item3, and item4".

Parameters

- **items** (*list*) – List of string items to join.
- **separator** (*str*, *optional*) – Item separator. Defaults to ', '.
- **last_separator** (*str*, *optional*) – Last item separator. Defaults to ' and '.

Returns Joined string.

Return type str

Example

```
>>> series_phrase_serial(['apples', 'bananas', 'peaches'])
'apples, bananas, and peaches'
```

New in version 3.0.0.

`pydash.strings.slugify(text, separator='-')`

Convert *text* into an ASCII slug which can be used safely in URLs. Incoming *text* is converted to unicode and normalized using the NFKD form. This results in some accented characters being converted to their ASCII “equivalent” (e.g. é is converted to e). Leading and trailing whitespace is trimmed and any remaining whitespace or other special characters without an ASCII equivalent are replaced with -.

Parameters

- **text** (*str*) – String to slugify.
- **separator** (*str*, *optional*) – Separator to use. Defaults to '- '.

Returns Slugified string.

Return type str

Example

```
>>> slugify('This is a slug.') == 'this-is-a-slug'
True
>>> slugify('This is a slug.', '+') == 'this+is+a+slug'
True
```

New in version 3.0.0.

`pydash.strings.snake_case(text)`

Converts *text* to snake case.

Parameters *text* (*str*) – String to convert.

Returns String converted to snake case.

Return type *str*

Example

```
>>> snake_case('This is Snake Case!')
'this_is_snake_case'
```

New in version 1.1.0.

Changed in version 4.0.0: Removed alias `underscore_case`.

`pydash.strings.split(text, separator=<pydash.helpers.NoValue object>)`

Splits *text* on *separator*. If *separator* not provided, then *text* is split on whitespace. If *separator* is falsey, then *text* is split on every character.

Parameters

- **text** (*str*) – String to explode.
- **separator** (*str*, *optional*) – Separator string to split on. Defaults to `NoValue`.

Returns Split string.

Return type *list*

Example

```
>>> split('one potato, two potatoes, three potatoes, four!')
['one', 'potato,', 'two', 'potatoes,', 'three', 'potatoes,', 'four!']
>>> split('one potato, two potatoes, three potatoes, four!', ',')
['one potato', ' two potatoes', ' three potatoes', ' four!']
```

New in version 2.0.0.

Changed in version 3.0.0: Changed *separator* default to `NoValue` and supported splitting on whitespace by default.

Changed in version 4.0.0: Removed alias `explode`.

`pydash.strings.start_case(text)`

Convert *text* to start case.

Parameters *text* (*str*) – String to convert.

Returns String converted to start case.

Return type str

Example

```
>>> start_case("fooBar")
'Foo Bar'
```

New in version 3.1.0.

`pydash.strings.starts_with(text, target, position=0)`

Checks if *text* starts with a given target string.

Parameters

- **text** (*str*) – String to check.
- **target** (*str*) – String to check for.
- **position** (*int*, *optional*) – Position to search from. Defaults to beginning of *text*.

Returns Whether *text* starts with *target*.

Return type bool

Example

```
>>> starts_with('abcdef', 'a')
True
>>> starts_with('abcdef', 'b')
False
>>> starts_with('abcdef', 'a', 1)
False
```

New in version 1.1.0.

`pydash.strings.strip_tags(text)`

Removes all HTML tags from *text*.

Parameters **text** (*str*) – String to strip.

Returns String without HTML tags.

Return type str

Example

```
>>> strip_tags('<a href="#">Some link</a>')
'Some link'
```

New in version 3.0.0.

`pydash.strings.substr_left(text, subtext)`

Searches *text* from left-to-right for *subtext* and returns a substring consisting of the characters in *text* that are to the left of *subtext* or all string if no match found.

Parameters

- **text** (*str*) – String to partition.
- **subtext** (*str*) – String to search for.

Returns Substring to left of *subtext*.

Return type str

Example

```
>>> substr_left('abcdefcdg', 'cd')
'ab'
```

New in version 3.0.0.

`pydash.strings.substr_left_end(text, subtext)`

Searches *text* from right-to-left for *subtext* and returns a substring consisting of the characters in *text* that are to the left of *subtext* or all string if no match found.

Parameters

- **text** (*str*) – String to partition.
- **subtext** (*str*) – String to search for.

Returns Substring to left of *subtext*.

Return type str

Example

```
>>> substr_left_end('abcdefcdg', 'cd')
'abcdef'
```

New in version 3.0.0.

`pydash.strings.substr_right(text, subtext)`

Searches *text* from right-to-left for *subtext* and returns a substring consisting of the characters in *text* that are to the right of *subtext* or all string if no match found.

Parameters

- **text** (*str*) – String to partition.
- **subtext** (*str*) – String to search for.

Returns Substring to right of *subtext*.

Return type str

Example

```
>>> substr_right('abcdefcdg', 'cd')
'efcdg'
```

New in version 3.0.0.

`pydash.strings.substr_right_end(text, subtext)`

Searches *text* from left-to-right for *subtext* and returns a substring consisting of the characters in *text* that are to the right of *subtext* or all string if no match found.

Parameters

- **text** (*str*) – String to partition.
- **subtext** (*str*) – String to search for.

Returns Substring to right of *subtext*.

Return type `str`

Example

```
>>> substr_right_end('abcdefcdg', 'cd')
'g'
```

New in version 3.0.0.

`pydash.strings.successor(char)`

Return the successor character of *char*.

Parameters **char** (*str*) – Character to find the successor of.

Returns Successor character.

Return type `str`

Example

```
>>> successor('b')
'c'
>>> successor('B')
'C'
>>> successor('2')
'3'
```

New in version 3.0.0.

`pydash.strings.surround(text, wrapper)`

Surround a string with another string.

Parameters

- **text** (*str*) – String to surround with *wrapper*.
- **wrapper** (*str*) – String by which *text* is to be surrounded.

Returns Surrounded string.

Return type `str`

Example

```
>>> surround('abc', '')
'abc'
>>> surround('abc', '!')
'!abc!'
```

New in version 2.4.0.

`pydash.strings.swap_case(text)`

Swap case of *text* characters.

Parameters *text* (*str*) – String to swap case.

Returns String with swapped case.

Return type *str*

Example

```
>>> swap_case('aBcDeF')
'AbCdEf'
```

New in version 3.0.0.

`pydash.strings.title_case(text)`

Convert *text* to title case.

Parameters *text* (*str*) – String to convert.

Returns String converted to title case.

Return type *str*

Example

```
>>> title_case("bob's shop")
"Bob's Shop"
```

New in version 3.0.0.

`pydash.strings.to_lower(text)`

Converts the given *text* to lower text.

Parameters *text* (*str*) – String to convert.

Returns String converted to lower case.

Return type *str*

Example

```
>>> to_lower('--Foo-Bar--')
'--foo-bar--'
>>> to_lower('fooBar')
'foobar'
>>> to_lower('__FOO_BAR__')
'__foo_bar__'
```

New in version 4.0.0.

`pydash.strings.to_upper(text)`

Converts the given `text` to upper text.

Parameters `text` (*str*) – String to convert.

Returns String converted to upper case.

Return type `str`

Example

```
>>> to_upper('--Foo-Bar--')
'--FOO-BAR--'
>>> to_upper('fooBar')
'FOOBAR'
>>> to_upper('__FOO_BAR__')
'__FOO_BAR__'
```

New in version 4.0.0.

`pydash.strings.trim(text, chars=None)`

Removes leading and trailing whitespace or specified characters from `text`.

Parameters

- **text** (*str*) – String to trim.
- **chars** (*str*, *optional*) – Specific characters to remove.

Returns Trimmed string.

Return type `str`

Example

```
>>> trim('  abc efg\r\n ')
'abc efg'
```

New in version 1.1.0.

`pydash.strings.trim_end(text, chars=None)`

Removes trailing whitespace or specified characters from `text`.

Parameters

- **text** (*str*) – String to trim.
- **chars** (*str*, *optional*) – Specific characters to remove.

Returns Trimmed string.

Return type `str`

Example

```
>>> trim_end('  abc efg\r\n ')
'  abc efg'
```

New in version 1.1.0.

Changed in version 4.0.0: Renamed from `trim_right` to `trim_end`.

`pydash.strings.trim_start(text, chars=None)`

Removes leading whitespace or specified characters from *text*.

Parameters

- **text** (*str*) – String to trim.
- **chars** (*str*, *optional*) – Specific characters to remove.

Returns Trimmed string.

Return type `str`

Example

```
>>> trim_start(' abc efg\r\n ')
'abc efg\r\n '
```

New in version 1.1.0.

Changed in version 4.0.0: Renamed from `trim_left` to `trim_start`.

`pydash.strings.truncate(text, length=30, omission='...', separator=None)`

Truncates *text* if it is longer than the given maximum string length. The last characters of the truncated string are replaced with the omission string which defaults to `...`

Parameters

- **text** (*str*) – String to truncate.
- **length** (*int*, *optional*) – Maximum string length. Defaults to 30.
- **omission** (*str*, *optional*) – String to indicate text is omitted.
- **separator** (*mixed*, *optional*) – Separator pattern to truncate to.

Returns Truncated string.

Return type `str`

Example

```
>>> truncate('hello world', 5)
'he...'
>>> truncate('hello world', 5, '..')
'hel..'
>>> truncate('hello world', 10)
'hello w...'
>>> truncate('hello world', 10, separator=' ')
'hello...'
```

New in version 1.1.0.

Changed in version 4.0.0: Removed alias `trunc`.

`pydash.strings.unescape(text)`

The inverse of `escape()`. This method converts the HTML entities `&`, `<`, `>`, `"`, `'`, and ``` in `text` to their corresponding characters.

Parameters `text` (*str*) – String to unescape.

Returns HTML unescaped string.

Return type `str`

Example

```
>>> results = unescape('&quot;1 &gt; 2 &amp;&amp; 3 &lt; 4&quot;')
>>> results == '"1 > 2 && 3 < 4"'
True
```

New in version 1.0.0.

Changed in version 1.1.0: Moved to `pydash.strings`.

`pydash.strings.unquote(text, quote_char="")`

Unquote `text` by removing `quote_char` if `text` begins and ends with it.

Parameters `text` (*str*) – String to unquote.

Returns Unquoted string.

Return type `str`

Example

```
>>> unquote('"abc"')
'abc'
>>> unquote('"abc"', '#')
'"abc"'
>>> unquote('#abc', '#')
'#abc'
>>> unquote('#abc#', '#')
'abc'
```

New in version 3.0.0.

`pydash.strings.upper_case(text)`

Converts string to upper case, as space separated words.

Parameters `text` (*str*) – String to be converted to uppercase.

Returns String converted to uppercase, as space separated words.

Return type `str`

Example

```
>>> upper_case('--foo-bar--')
'FOO BAR'
>>> upper_case('fooBar')
'FOO BAR'
```

```
>>> upper_case('/?*Foo10/;"B*Ar')
'FOO 10 B AR'
```

New in version 4.0.0.

`pydash.strings.upper_first(text)`

Converts the first character of string to upper case.

Parameters `text` (*str*) – String passed in by the user.

Returns String in which the first character is converted to upper case.

Return type `str`

Example

```
>>> upper_first('fred')
'Fred'
>>> upper_first('foo bar')
'Foo bar'
>>> upper_first('lfoobar')
'lfoobar'
>>> upper_first(';foobar')
';foobar'
```

New in version 4.0.0.

`pydash.strings.url(*paths, **params)`

Combines a series of URL paths into a single URL. Optionally, pass in keyword arguments to append query parameters.

Parameters `paths` (*str*) – URL paths to combine.

Keyword Arguments `params` (*str, optional*) – Query parameters.

Returns URL string.

Return type `str`

Example

```
>>> link = url('a', 'b', ['c', 'd'], '/', q='X', y='Z')
>>> path, params = link.split('?')
>>> path == 'a/b/c/d/'
True
>>> set(params.split('&')) == set(['q=X', 'y=Z'])
True
```

New in version 2.2.0.

`pydash.strings.words(text, pattern=None)`

Return list of words contained in `text`.

Parameters

- **text** (*str*) – String to split.
- **pattern** (*str, optional*) – Custom pattern to split words on. Defaults to `None`.

Returns List of words.

Return type list

Example

```
>>> words('a b, c; d-e')
['a', 'b', 'c', 'd', 'e']
>>> words('fred, barney, & pebbles', '/[^, ]+/g')
['fred', 'barney', '&', 'pebbles']
```

New in version 2.0.0.

Changed in version 3.2.0: Added *pattern* argument.

Changed in version 3.2.0: Improved matching for one character words.

Utilities

Utility functions.

New in version 1.0.0.

`pydash.utilities.attempt(func, *args, **kwargs)`

Attempts to execute *func*, returning either the result or the caught error object.

Parameters *func* (*function*) – The function to attempt.

Returns Returns the *func* result or error object.

Return type mixed

Example

```
>>> results = attempt(lambda x: x/0, 1)
>>> assert isinstance(results, ZeroDivisionError)
```

New in version 1.1.0.

`pydash.utilities.cond(*pairs)`

Creates a function that iterates over *pairs* and invokes the corresponding function of the first predicate to return truthy.

Parameters *pairs* (*list*) – A list of predicate-function pairs.

Returns Returns the new composite function.

Return type function

Example

```
>>> func = cond([matches({'a': 1}), constant('matches A')])
>>> func({'a': 1, 'b': 2})
'matches A'
```

New in version 4.0.0.

`pydash.utilities.conforms` (*source*)

Creates a function that invokes the predicate properties of *source* with the corresponding property values of a given object, returning `True` if all predicates return truthy, else `False`.

Parameters *source* (*dict*/*list*) – The object of property predicates to conform to.

Returns Returns the new spec function.

Return type function

Example

```
>>> func = conforms({'b': lambda n: n > 1})
>>> func({'b': 2})
True
>>> func({'b': 0})
False
>>> func = conforms([lambda n: n > 1, lambda n: n == 0])
>>> func([2, 0])
True
>>> func([0, 0])
False
```

New in version 4.0.0.

`pydash.utilities.conforms_to` (*obj*, *source*)

Checks if *obj* conforms to *source* by invoking the predicate properties of *source* with the corresponding property values of *obj*.

Parameters

- **obj** (*dict*/*list*) – The object to inspect.
- **source** (*dict*/*list*) – The object of property predicates to conform to.

Example

```
>>> conforms_to({'b': 2}, {'b': lambda n: n > 1})
True
>>> conforms_to({'b': 0}, {'b': lambda n: n > 1})
False
>>> conforms_to([2, 0], [lambda n: n > 1, lambda n: n == 0])
True
>>> conforms_to([0, 0], [lambda n: n > 1, lambda n: n == 0])
False
```

New in version 4.0.0.

`pydash.utilities.constant` (*value*)

Creates a function that returns *value*.

Parameters *value* (*mixed*) – Constant value to return.

Returns Function that always returns *value*.

Return type function

Example

```
>>> pi = constant(3.14)
>>> pi() == 3.14
True
```

New in version 1.0.0.

Changed in version 4.0.0: Returned function ignores arguments instead of raising exception.

`pydash.utilities.default_to(value, default_value)`

Checks `value` to determine whether a default value should be returned in its place. The `default_value` is returned if `value` is `None`.

Parameters

- **value** (*mixed*) – Value passed in by the user.
- **default_value** (*mixed*) – Default value passed in by the user.

Returns

Returns **value** if **value** is given otherwise returns `default_value`.

Return type `mixed`

Example

```
>>> default_to(1, 10)
1
>>> default_to(None, 10)
10
```

New in version 4.0.0.

`pydash.utilities.identity(*args)`

Return the first argument provided to it.

Parameters ***args** (*mixed*) – Arguments.

Returns First argument or `None`.

Return type `mixed`

Example

```
>>> identity(1)
1
>>> identity(1, 2, 3)
1
>>> identity() is None
True
```

New in version 1.0.0.

`pydash.utilities.iteratee(func)`

Return a pydash style iteratee. If `func` is a property name the created iteratee will return the property value for a given element. If `func` is an object the created iteratee will return `True` for elements that contain the equivalent object properties, otherwise it will return `False`.

Parameters `func` (*mixed*) – Object to create iteratee function from.

Returns Iteratee function.

Return type function

Example

```
>>> get_data = iteratee('data')
>>> get_data({'data': [1, 2, 3]})
[1, 2, 3]
>>> is_active = iteratee({'active': True})
>>> is_active({'active': True})
True
>>> is_active({'active': 0})
False
>>> iteratee(['a', 5])({'a': 5})
True
>>> iteratee(['a.b'])({'a.b': 5})
5
>>> iteratee('a.b')({'a': {'b': 5}})
5
>>> iteratee(lambda a, b: a + b)(1, 2)
3
>>> ident = iteratee(None)
>>> ident('a')
'a'
>>> ident(1, 2, 3)
1
```

New in version 1.0.0.

Changed in version 2.0.0: Renamed `create_iteratee()` to `iteratee()`.

Changed in version 3.0.0: Made pluck style iteratee support deep property access.

Changed in version 3.1.0: - Added support for shallow pluck style property access via single item list/tuple. - Added support for matches property style iteratee via two item list/tuple.

Changed in version 4.0.0: Removed alias `callback`.

`pydash.utilities.matches` (*source*)

Creates a `pydash.collections.where()` style predicate function which performs a deep comparison between a given object and the *source* object, returning `True` if the given object has equivalent property values, else `False`.

Parameters `source` (*dict*) – Source object used for comparison.

Returns

Function that compares an object to *source* and returns whether the two objects contain the same items.

Return type function

Example

```
>>> matches({'a': {'b': 2}})({'a': {'b': 2, 'c': 3}})
True
>>> matches({'a': 1})({'b': 2, 'a': 1})
True
>>> matches({'a': 1})({'b': 2, 'a': 2})
False
```

New in version 1.0.0.

Changed in version 3.0.0: Use `pydash.predicates.is_match()` as matching function.

`pydash.utilities.matches_property(key, value)`

Creates a function that compares the property value of *key* on a given object to *value*.

Parameters

- **key** (*str*) – Object key to match against.
- **value** (*mixed*) – Value to compare to.

Returns

Function that compares *value* to an object's *key* and returns whether they are equal.

Return type function

Example

```
>>> matches_property('a', 1)({'a': 1, 'b': 2})
True
>>> matches_property(0, 1)([1, 2, 3])
True
>>> matches_property('a', 2)({'a': 1, 'b': 2})
False
```

New in version 3.1.0.

`pydash.utilities.memoize(func, resolver=None)`

Creates a function that memoizes the result of *func*. If *resolver* is provided it will be used to determine the cache key for storing the result based on the arguments provided to the memoized function. By default, all arguments provided to the memoized function are used as the cache key. The result cache is exposed as the `cache` property on the memoized function.

Parameters

- **func** (*function*) – Function to memoize.
- **resolver** (*function, optional*) – Function that returns the cache key to use.

Returns Memoized function.

Return type function

Example

```

>>> ident = memoize(identity)
>>> ident(1)
1
>>> ident.cache['(1,){}'] == 1
True
>>> ident(1, 2, 3)
1
>>> ident.cache['(1, 2, 3){}'] == 1
True

```

New in version 1.0.0.

`pydash.utilities.method(path, *args, **kwargs)`

Creates a function that invokes the method at *path* on a given object. Any additional arguments are provided to the invoked method.

Parameters

- **path** (*str*) – Object path of method to invoke.
- ***args** (*mixed*) – Global arguments to apply to method when invoked.
- ****kwargs** (*mixed*) – Global keyword argument to apply to method when invoked.

Returns Function that invokes method located at path for object.

Return type function

Example

```

>>> obj = {'a': {'b': [None, lambda x: x]}}
>>> echo = method('a.b.1')
>>> echo(obj, 1) == 1
True
>>> echo(obj, 'one') == 'one'
True

```

New in version 3.3.0.

`pydash.utilities.method_of(obj, *args, **kwargs)`

The opposite of `method()`. This method creates a function that invokes the method at a given path on object. Any additional arguments are provided to the invoked method.

Parameters

- **obj** (*mixed*) – The object to query.
- ***args** (*mixed*) – Global arguments to apply to method when invoked.
- ****kwargs** (*mixed*) – Global keyword argument to apply to method when invoked.

Returns Function that invokes method located at path for object.

Return type function

Example

```
>>> obj = {'a': {'b': [None, lambda x: x]}}
>>> dispatch = method_of(obj)
>>> dispatch('a.b.1', 1) == 1
True
>>> dispatch('a.b.1', 'one') == 'one'
True
```

New in version 3.3.0.

`pydash.utilities.noop(*args, **kwargs)`

A no-operation function.

New in version 1.0.0.

`pydash.utilities.nth_arg(pos=0)`

Creates a function that gets the argument at index *n*. If *n* is negative, the *nth* argument from the end is returned.

Parameters *pos* (*int*) – The index of the argument to return.

Returns Returns the new pass-thru function.

Return type function

Example

```
>>> func = nth_arg(1)
>>> func(11, 22, 33, 44)
22
>>> func = nth_arg(-1)
>>> func(11, 22, 33, 44)
44
```

New in version 4.0.0.

`pydash.utilities.now()`

Return the number of milliseconds that have elapsed since the Unix epoch (1 January 1970 00:00:00 UTC).

Returns Milliseconds since Unix epoch.

Return type *int*

New in version 1.0.0.

Changed in version 3.0.0: Use `datetime` module for calculating elapsed time.

`pydash.utilities.over(funcs)`

Creates a function that invokes all functions in *funcs* with the arguments it receives and returns their results.

Parameters *funcs* (*list*) – List of functions to be invoked.

Returns Returns the new pass-thru function.

Return type function

Example

```
>>> func = over([max, min])
>>> func(1, 2, 3, 4)
[4, 1]
```

New in version 4.0.0.

`pydash.utilities.over_every(funcs)`

Creates a function that checks if all of the functions in *funcs* return truthy when invoked with the arguments it receives.

Parameters *funcs* (*list*) – List of functions to be invoked.

Returns Returns the new pass-thru function.

Return type function

Example

```
>>> func = over_every([bool, lambda x: x is not None])
>>> func(1)
True
```

New in version 4.0.0.

`pydash.utilities.over_some(funcs)`

Creates a function that checks if any of the functions in *funcs* return truthy when invoked with the arguments it receives.

Parameters *funcs* (*list*) – List of functions to be invoked.

Returns Returns the new pass-thru function.

Return type function

Example

```
>>> func = over_some([bool, lambda x: x is None])
>>> func(1)
True
```

New in version 4.0.0.

`pydash.utilities.property_(path)`

Creates a function that returns the value at path of a given object.

Parameters *path* (*str/list*) – Path value to fetch from object.

Returns Function that returns object's path value.

Return type function

Example

```
>>> get_data = property_('data')
>>> get_data({'data': 1})
1
>>> get_data({}) is None
True
>>> get_first = property_(0)
>>> get_first([1, 2, 3])
1
```

New in version 1.0.0.

Changed in version 4.0.1: Made property accessor work with deep path strings.

`pydash.utilities.property_of(obj)`

The inverse of `property_()`. This method creates a function that returns the key value of a given key on *obj*.

Parameters `obj` (*dict*/*list*) – Object to fetch values from.

Returns Function that returns object's key value.

Return type function

Example

```
>>> getter = property_of({'a': 1, 'b': 2, 'c': 3})
>>> getter('a')
1
>>> getter('b')
2
>>> getter('x') is None
True
```

New in version 3.0.0.

Changed in version 4.0.0: Removed alias `prop_of`.

`pydash.utilities.random(start=0, stop=1, floating=False)`

Produces a random number between *start* and *stop* (inclusive). If only one argument is provided a number between 0 and the given number will be returned. If *floating* is truthy or either *start* or *stop* are floats a floating-point number will be returned instead of an integer.

Parameters

- **start** (*int*) – Minimum value.
- **stop** (*int*) – Maximum value.
- **floating** (*bool*, *optional*) – Whether to force random value to float. Default is False.

Returns Random value.

Return type `int`/`float`

Example

```
>>> 0 <= random() <= 1
True
>>> 5 <= random(5, 10) <= 10
True
>>> isinstance(random(floating=True), float)
True
```

New in version 1.0.0.

`pydash.utilities.range_(*args)`

Creates a list of numbers (positive and/or negative) progressing from *start* up to but not including *end*. If *start* is less than *stop*, a zero-length range is created unless a negative *step* is specified.

Parameters

- **start** (*int*, *optional*) – Integer to start with. Defaults to 0.
- **stop** (*int*) – Integer to stop at.
- **step** (*int*, *optional*) – The value to increment or decrement by. Defaults to 1.

Yields *int* – Next integer in range.

Example

```
>>> list(range_(5))
[0, 1, 2, 3, 4]
>>> list(range_(1, 4))
[1, 2, 3]
>>> list(range_(0, 6, 2))
[0, 2, 4]
>>> list(range_(4, 1))
[4, 3, 2]
```

New in version 1.0.0.

Changed in version 1.1.0: Moved to `pydash.utilities`.

Changed in version 3.0.0: Return generator instead of list.

Changed in version 4.0.0: Support decrementing when start argument is greater than stop argument.

`pydash.utilities.range_right(*args)`

Similar to `range_()`, except that it populates the values in descending order.

Parameters

- **start** (*int*, *optional*) – Integer to start with. Defaults to 0.
- **stop** (*int*) – Integer to stop at.
- **step** (*int*, *optional*) – The value to increment or decrement by. Defaults to 1 if *start* < *stop* else -1.

Yields *int* – Next integer in range.

Example

```
>>> list(range_right(5))
[4, 3, 2, 1, 0]
>>> list(range_right(1, 4))
[3, 2, 1]
>>> list(range_right(0, 6, 2))
[4, 2, 0]
```

New in version 4.0.0.

`pydash.utilities.result(obj, key, default=None)`

Return the value of property *key* on *obj*. If *key* value is a function it will be invoked and its result returned, else the property value is returned. If *obj* is falsey then *default* is returned.

Parameters

- **obj** (*list* | *dict*) – Object to retrieve result from.

- **key** (*mixed*) – Key or index to get result from.
- **default** (*mixed, optional*) – Default value to return if *obj* is falsey. Defaults to None.

Returns Result of `obj[key]` or None.

Return type mixed

Example

```
>>> result({'a': 1, 'b': lambda: 2}, 'a')
1
>>> result({'a': 1, 'b': lambda: 2}, 'b')
2
>>> result({'a': 1, 'b': lambda: 2}, 'c') is None
True
>>> result({'a': 1, 'b': lambda: 2}, 'c', default=False)
False
```

New in version 1.0.0.

Changed in version 2.0.0: Added default argument.

`pydash.utilities.stub_list()`

Returns empty “list”.

Returns Empty list.

Return type list

Example

```
>>> stub_list()
[]
```

New in version 4.0.0.

`pydash.utilities.stub_dict()`

Returns empty “dict”.

Returns Empty dict.

Return type dict

Example

```
>>> stub_dict()
{}
```

New in version 4.0.0.

`pydash.utilities.stub_false()`

Returns False.

Returns False

Return type bool

Example

```
>>> stub_false()
False
```

New in version 4.0.0.

`pydash.utilities.stub_string()`

Returns an empty string.

Returns Empty string

Return type str

Example

```
>>> stub_string()
''
```

New in version 4.0.0.

`pydash.utilities.stub_true()`

Returns True.

Returns True

Return type bool

Example

```
>>> stub_true()
True
```

New in version 4.0.0.

`pydash.utilities.times(n, iteratee=<function identity>)`

Executes the *iteratee* *n* times, returning a list of the results of each *iteratee* execution. The *iteratee* is invoked with one argument: (*index*).

Parameters

- **n** (*int*) – Number of times to execute *iteratee*.
- **iteratee** (*function*) – Function to execute.

Returns A list of results from calling *iteratee*.

Return type list

Example

```
>>> times(5, lambda i: i)
[0, 1, 2, 3, 4]
```

New in version 1.0.0.

Changed in version 3.0.0: Reordered arguments to make *iteratee* first.

Changed in version 4.0.0: Re-reordered arguments to make *iteratee* last argument. Added functionality for handling *iteratee* with zero positional arguments.

`pydash.utilities.to_path(value)`

Converts values to a property path array.

Parameters `value` (*mixed*) – Value to convert.

Returns Returns the new property path array.

Return type list

Example

```
>>> to_path('a.b.c')
['a', 'b', 'c']
>>> to_path('a[0].b.c')
['a', 0, 'b', 'c']
>>> to_path('a[0][1][2].b.c')
['a', 0, 1, 2, 'b', 'c']
```

New in version 4.0.0.

`pydash.utilities.unique_id(prefix=None)`

Generates a unique ID. If *prefix* is provided the ID will be appended to it.

Parameters `prefix` (*str, optional*) – String prefix to prepend to ID value.

Returns ID value.

Return type str

Example

```
>>> unique_id()
'1'
>>> unique_id('id_')
'id_2'
>>> unique_id()
'3'
```

New in version 1.0.0.

License

The MIT License (MIT)

Copyright (c) 2014 Derrick Gilland

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

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

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

Versioning

This project follows [Semantic Versioning](#) with the following caveats:

- Only the public API (i.e. the objects imported into the `pydash` module) will maintain backwards compatibility between MINOR version bumps.
- Objects within any other parts of the library are not guaranteed to not break between MINOR version bumps.

With that in mind, it is recommended to only use or import objects from the main module, `pydash`.

Changelog

v4.0.3 (2017-04-20)

New Features

- None

Bug Fixes

- Fix regression in `get` where `list` and `dict` objects had attributes returned when a key was missing but the key corresponded to an attribute name. For example, `pydash.get({}, 'update')` would return `{}.update()` instead of `None`. Previous behavior was that only item-access was allowed for `list` and `dict` which has been restored.
- Fix regression in `invoke/invoke_map` where non-attributes could be invoked. For example, `pydash.invoke({'items': lambda: 1}, 'items')` would return `1` instead of `dict_items([('a', 'items')])`. Previous behavior was that only attribute methods could be invoked which has now been restored.

v4.0.2 (2017-04-04)

New Features

- None

Bug Fixes

- Fix regression in `intersection`, `intersection_by`, and `intersection_with` introduced in v4.0.0 where the a single argument supplied to `intersection` should return the same argument value instead of an empty list.

Backwards-Incompatibilities

- None

v4.0.1 (2017-04-04)

New Features

- Make `property_` work with deep path strings.

Bug Fixes

- Revert removal of `deep_pluck` and rename to `pluck`. Previously, `deep_pluck` was removed and `map_` was recommended as a replacement. However, `deep_pluck` (now defined as `pluck`) functionality is not supported by `map_` so the removal `pluck` was reverted.

Backwards-Incompatibilities

- Remove `property_deep` (use `property_`).

v4.0.0 (2017-04-03)

New Features

- Add `assign_with`.
- Add `clamp`.
- Add `clone_deep_with`.
- Add `clone_with`.
- Add `cond`. Thanks [bharadwajyarlagadda!](#)
- Add `conforms`.
- Add `conforms_to`.
- Add `default_to`. Thanks [bharadwajyarlagadda!](#)
- Add `difference_by`.
- Add `difference_with`.
- Add `divide`. Thanks [bharadwajyarlagadda!](#)
- Add `eq`. Thanks [bharadwajyarlagadda!](#)
- Add `flat_map`.
- Add `flat_map_deep`.
- Add `flat_map_depth`.
- Add `flatten_depth`.
- Add `flip`. Thanks [bharadwajyarlagadda!](#)
- Add `from_pairs`. Thanks [bharadwajyarlagadda!](#)
- Add `intersection_by`.
- Add `intersection_with`.
- Add `invert_by`.
- Add `invoke_map`.
- Add `is_equal_with`. Thanks [bharadwajyarlagadda!](#)
- Add `is_match_with`.
- Add `is_set`. Thanks [bharadwajyarlagadda!](#)
- Add `lower_case`. Thanks [bharadwajyarlagadda!](#)
- Add `lower_first`. Thanks [bharadwajyarlagadda!](#)
- Add `max_by`.
- Add `mean_by`.
- Add `merge_with`.

- Add `min_by`.
- Add `multiply`. Thanks [bharadwajyarlagadda!](#)
- Add `nth`. Thanks [bharadwajyarlagadda!](#)
- Add `nth_arg`. Thanks [bharadwajyarlagadda!](#)
- Add `omit_by`.
- Add `over`. Thanks [bharadwajyarlagadda!](#)
- Add `over_every`. Thanks [bharadwajyarlagadda!](#)
- Add `over_some`. Thanks [bharadwajyarlagadda!](#)
- Add `pick_by`.
- Add `pull_all`. Thanks [bharadwajyarlagadda!](#)
- Add `pull_all_by`.
- Add `pull_all_with`.
- Add `range_right`. Thanks [bharadwajyarlagadda!](#)
- Add `sample_size`. Thanks [bharadwajyarlagadda!](#)
- Add `set_with`.
- Add `sorted_index_by`.
- Add `sorted_index_of`. Thanks [bharadwajyarlagadda!](#)
- Add `sorted_last_index_by`.
- Add `sorted_last_index_of`.
- Add `sorted_uniq`. Thanks [bharadwajyarlagadda!](#)
- Add `sorted_uniq_by`.
- Add `stub_list`. Thanks [bharadwajyarlagadda!](#)
- Add `stub_dict`. Thanks [bharadwajyarlagadda!](#)
- Add `stub_false`. Thanks [bharadwajyarlagadda!](#)
- Add `stub_string`. Thanks [bharadwajyarlagadda!](#)
- Add `stub_true`. Thanks [bharadwajyarlagadda!](#)
- Add `subtract`. Thanks [bharadwajyarlagadda!](#)
- Add `sum_by`.
- Add `to_integer`.
- Add `to_lower`. Thanks [bharadwajyarlagadda!](#)
- Add `to_path`. Thanks [bharadwajyarlagadda!](#)
- Add `to_upper`. Thanks [bharadwajyarlagadda!](#)
- Add `unary`.
- Add `union_by`. Thanks [bharadwajyarlagadda!](#)
- Add `union_with`. Thanks [bharadwajyarlagadda!](#)
- Add `uniq_by`.

- Add `uniq_with`.
- Add `unset`.
- Add `update`.
- Add `update_with`.
- Add `upper_case`. Thanks [bharadwajyarlagadda!](#)
- Add `upper_first`. Thanks [bharadwajyarlagadda!](#)
- Add `xor_by`.
- Add `xor_with`.
- Add `zip_object_deep`.
- Make function returned by `constant` ignore extra arguments when called.
- Make `get` support attribute access within path.
- Make `iteratee` treat an integer argument as a string path (i.e. `iteratee(1)` is equivalent to `iteratee('1')` for creating a path accessor function).
- Make `intersection` work with unhashable types.
- Make `range_` support decrementing when `start` argument is greater than `stop` argument.
- Make `xor` maintain sort order of supplied arguments.

Bug Fixes

- Fix `find_last_key` so that it iterates over object in reverse.

Backwards-Incompatibilities

- Make `add` only support two argument addition. (**breaking change**)
- Make `difference` return duplicate values from first argument and maintain sort order. (**breaking change**)
- Make `invoke` work on objects instead of collections. Use `invoke_map` for collections. (**breaking change**)
- Make `set_` support mixed `list/dict` defaults within a single object based on whether key or index path substrings used. (**breaking change**)
- Make `set_` modify object in place. (**breaking change**)
- Only use `merge` callback result if result is not `None`. Previously, result from callback (if provided) was used unconditionally. (**breaking change**)
- Remove functions: (**breaking change**)
 - `deep_pluck` (no alternative) [**UPDATE:** `deep_pluck` functionality restored as `pluck` in v4.0.1]
 - `mapiter` (no alternative)
 - `pluck` (use `map_`)
 - `update_path` (use `update` or `update_with`)
 - `set_path` (use `set_` or `set_with`)
- Remove aliases: (**breaking change**)
 - `all_` (use `every`)

- any_ (use some)
- append (use push)
- average and avg (use mean or mean_by)
- callback (use iteratee)
- cat (use concat)
- collect (use map_)
- contains (use includes)
- curve (use round_)
- deep_get and get_path (use get)
- deep_has and has_path (use has)
- deep_prop (use property_deep)
- deep_set (use set_)
- detect and find_where (use find)
- each (use for_each)
- each_right (use for_each_right)
- escape_re (use escape_reg_exp)
- explode (use split)
- extend (use assign)
- first (use head)
- foldl (use reduce)
- foldr (use reduce_right)
- for_own (use for_each)
- for_own_right (use for_each_right)
- implode (use join)
- is_bool (use is_boolean)
- is_int (use is_integer)
- is_native (use is_builtin)
- is_num (use is_number)
- is_plain_object (use is_dict)
- is_re (use is_reg_exp)
- js_match (use reg_exp_js_match)
- js_replace (use reg_exp_js_replace)
- keys_in (use keys)
- moving_average and moving_avg (use moving_mean)
- object_ (use zip_object)
- pad_left (use pad_start)

- pad_right (use pad_end)
- pipe (use flow)
- pipe_right and compose (use flow_right)
- prop (use property_)
- prop_of (use property_of)
- pow_ (use power)
- re_replace (use reg_exp_replace)
- rest (use tail)
- select (use filter_)
- sigma (use std_deviation)
- sort_by_all and sort_by_order (use order_by)
- trim_left (use trim_start)
- trim_right (use trim_right)
- trunc (use truncate)
- underscore_case (use snake_case)
- unique (use uniq)
- values_in (use values)
- where (use filter_)

- **Rename functions: (breaking change)**

- deep_map_values to map_values_deep
- deep_property to property_deep
- include to includes
- index_by to key_by
- mod_args to over_args
- moving_average to moving_mean
- pairs to to_pairs

- **Remove callback argument from: (breaking change)**

- assign. Moved to assign_with.
- clone and clone_deep. Moved to clone_with and clone_deep_with.
- is_match. Moved to is_match_with.
- max_ and min_. Moved to max_by and min_by.
- omit. Moved to omit_by.
- pick. Moved to pick_by.
- sorted_index. Moved to sorted_index_by.
- sum_. Moved to sum_by.
- uniq/unique. Moved to uniq_by.

- Renamed callback argument to predicate: **(breaking change)**

- drop_right_while
- drop_while
- every
- filter_
- find
- find_key
- find_last
- find_index
- find_last_index
- find_last_key
- partition
- reject
- remove
- some
- take_right_while
- take_while

- Renamed callback argument to iteratee: **(breaking change)**

- count_by
- duplicates
- for_each
- for_each_right
- for_in
- for_in_right
- group_by
- key_by
- map_
- map_keys
- map_values
- map_values_deep
- mapcat
- median
- reduce_
- reduce_right
- reductions
- reductions_right

- `sort_by`
- `times`
- `transform`
- `unzip_with`
- `zip_with`
- `zscore`

- Rename `comparison` argument in `sort` to `comparator`.
- Rename `index` and `how_many` arguments in `splice` to `start` and `count`.
- Remove `multivalue` argument from `invert`. Feature moved to `invert_by`. (**breaking change**)

v3.4.8 (2017-01-05)

- Make internal function inspection methods work with Python 3 annotations. Thanks [tgriesser](#)!

v3.4.7 (2016-11-01)

- Fix bug in `get` where an iterable default was iterated over instead of being returned when an object path wasn't found. Thanks [urbnjamessmi](#)!

v3.4.6 (2016-10-31)

- Fix bug in `get` where casting a string key to integer resulted in an uncaught exception instead of the default value being returned instead. Thanks [urbnjamessmi](#)!

v3.4.5 (2016-10-16)

- Add optional `default` parameter to `min_` and `max_` functions that is used when provided iterable is empty.
- Fix bug in `is_match` where comparison between an empty `source` argument returned `None` instead of `True`.

v3.4.4 (2016-09-06)

- Shallow copy each source in `assign/extend` instead of deep copying.
- Call `copy.deepcopy` in `merge` instead of the more resource intensive `clone_deep`.

v3.4.3 (2016-04-07)

- Fix minor issue in deep path string parsing so that list indexing in paths can be specified as `foo[0][1].bar` instead of `foo.[0].[1].bar`. Both formats are now supported.

v3.4.2 (2016-03-24)

- Fix bug in `start_case` where capitalized characters after the first character of a word were mistakenly cast to lower case.

v3.4.1 (2015-11-03)

- Fix Python 3.5, inspect, and pytest compatibility issue with `py_chaining` object when doctest run on `pydash.__init__.py`.

v3.4.0 (2015-09-22)

- Optimize callback system for performance.
 - Explicitly store arg count on callback for `pydash` generated callbacks where the arg count is known. This avoids the costly `inspect.getargspec` call.
 - Eliminate usage of costly `guess_builtin_argcount` which parsed docstrings, and instead only ever pass a single argument to a builtin callback function.
- Optimize `get/set` so that regex parsing is only done when special characters are contained in the path key whereas before, all string paths were parsed.
- Optimize `is_builtin` by checking for `BuiltinFunctionType` instance and then using `dict` look up table instead of a `list` look up.
- Optimize `is_match` by replacing call to `has` with a `try/except` block.
- Optimize `push/append` by using a native loop instead of callback mapping.

v3.3.0 (2015-07-23)

- Add `ceil`.
- Add `defaults_deep`.
- Add `floor`.
- Add `get`.
- Add `gt`.
- Add `gte`.
- Add `is_iterable`.
- Add `lt`.
- Add `lte`.
- Add `map_keys`.
- Add `method`.
- Add `method_of`.
- Add `mod_args`.
- Add `set_`.
- Add `unzip_with`.

- Add `zip_with`.
- Make `add` support adding two numbers if passed in positionally.
- Make `get` main definition and `get_path` its alias.
- Make `set_main` definition and `deep_set` its alias.

v3.2.2 (2015-04-29)

- Catch `AttributeError` in `helpers.get_item` and return default value if set.

v3.2.1 (2015-04-29)

- Fix bug in `reduce_right` where collection was not reversed correctly.

v3.2.0 (2015-03-03)

- Add `sort_by_order` as alias of `sort_by_all`.
- Fix `is_match` to not compare `obj` and `source` types using `type` and instead use `isinstance` comparisons exclusively.
- Make `sort_by_all` accept an `orders` argument for specifying the sort order of each key via boolean `True` (for ascending) and `False` (for descending).
- Make `words` accept a `pattern` argument to override the default regex used for splitting words.
- Make `words` handle single character words better.

v3.1.0 (2015-02-28)

- Add `fill`.
- Add `in_range`.
- Add `matches_property`.
- Add `spread`.
- Add `start_case`.
- Make callbacks support `matches_property` style as `[key, value]` or `(key, value)`.
- Make callbacks support shallow property style callbacks as `[key]` or `(key,)`.

v3.0.0 (2015-02-25)

- Add `ary`.
- Add `chars`.
- Add `chop`.
- Add `chop_right`.
- Add `clean`.

- Add `commit` method to `chain` that returns a new chain with the computed `chain.value()` as the initial value of the chain.
- Add `count_substr`.
- Add `decapitalize`.
- Add `duplicates`.
- Add `has_substr`.
- Add `human_case`.
- Add `insert_substr`.
- Add `is_blank`.
- Add `is_bool` as alias of `is_boolean`.
- Add `is_builtin`, `is_native`.
- Add `is_dict` as alias of `is_plain_object`.
- Add `is_int` as alias of `is_integer`.
- Add `is_match`.
- Add `is_num` as alias of `is_number`.
- Add `is_tuple`.
- Add `join` as alias of `implode`.
- Add `lines`.
- Add `number_format`.
- Add `pascal_case`.
- Add `plant` method to `chain` that returns a cloned chain with a new initial value.
- Add `predecessor`.
- Add `property_of`, `prop_of`.
- Add `prune`.
- Add `re_replace`.
- Add `rearg`.
- Add `replace`.
- Add `run` as alias of `chain.value`.
- Add `separator_case`.
- Add `series_phrase`.
- Add `series_phrase_serial`.
- Add `slugify`.
- Add `sort_by_all`.
- Add `strip_tags`.
- Add `substr_left`.
- Add `substr_left_end`.

- Add `substr_right`.
- Add `substr_right_end`.
- Add `successor`.
- Add `swap_case`.
- Add `title_case`.
- Add `truncate` as alias of `trunc`.
- Add `to_boolean`.
- Add `to_dict`, `to_plain_object`.
- Add `to_number`.
- Add `underscore_case` as alias of `snake_case`.
- Add `unquote`.
- Fix `deep_has` to return `False` when `ValueError` raised during path checking.
- Fix `pad` so that it doesn't over pad beyond provided length.
- Fix `trunc`/`truncate` so that they handle texts shorter than the max string length correctly.
- Make the following functions work with empty strings and `None`: **(breaking change)** Thanks [k7sleeper](#)!
 - `camel_case`
 - `capitalize`
 - `chars`
 - `chop`
 - `chop_right`
 - `class_case`
 - `clean`
 - `count_substr`
 - `decapitalize`
 - `ends_with`
 - `join`
 - `js_replace`
 - `kebab_case`
 - `lines`
 - `quote`
 - `re_replace`
 - `replace`
 - `series_phrase`
 - `series_phrase_serial`
 - `starts_with`
 - `surround`

- Make callback invocation have better support for builtin functions and methods. Previously, if one wanted to pass a builtin function or method as a callback, it had to be wrapped in a lambda which limited the number of arguments that would be passed it. For example, `_.each([1, 2, 3], array.append)` would fail and would need to be converted to `_.each([1, 2, 3], lambda item: array.append(item))`. That is no longer the case as the non-wrapped method is now supported.
- Make `capitalize` accept `strict` argument to control whether to convert the rest of the string to lower case or not. Defaults to `True`.
- Make `chain` support late passing of initial value argument.
- Make `chain` not store computed value(). **(breaking change)**
- Make `drop`, `drop_right`, `take`, and `take_right` have default `n=1`.
- Make `is_indexed` return `True` for tuples.
- Make `partial` and `partial_right` accept keyword arguments.
- Make `pluck` style callbacks support deep paths. **(breaking change)**
- Make `re_replace` accept non-string arguments.
- Make `sort_by` accept `reverse` parameter.
- Make `splice` work with strings.
- Make `to_string` convert `None` to empty string. **(breaking change)**
- Move `arrays.join` to `strings.join`. **(breaking change)**
- Rename `join/implode`'s second parameter from `delimiter` to `separator`. **(breaking change)**
- Rename `split/explode`'s second parameter from `delimiter` to `separator`. **(breaking change)**
- Reorder function arguments for `after` from `(n, func)` to `(func, n)`. **(breaking change)**
- Reorder function arguments for `before` from `(n, func)` to `(func, n)`. **(breaking change)**
- Reorder function arguments for `times` from `(n, callback)` to `(callback, n)`. **(breaking change)**
- Reorder function arguments for `js_match` from `(reg_exp, text)` to `(text, reg_exp)`. **(breaking change)**
- Reorder function arguments for `js_replace` from `(reg_exp, text, repl)` to `(text, reg_exp, repl)`. **(breaking change)**
- Support iteration over class instance properties for non-list, non-dict, and non-iterable objects.

v2.4.2 (2015-02-03)

- Fix `remove` so that array is modified after callback iteration.

v2.4.1 (2015-01-11)

- Fix `kebab_case` so that it casts string to lower case.

v2.4.0 (2015-01-07)

- Add `ensure_ends_with`. Thanks [k7sleeper!](#)
- Add `ensure_starts_with`. Thanks [k7sleeper!](#)
- Add `quote`. Thanks [k7sleeper!](#)
- Add `surround`. Thanks [k7sleeper!](#)

v2.3.2 (2014-12-10)

- Fix `merge` and `assign/extend` so they apply `clone_deep` to source values before assigning to destination object.
- Make `merge` accept a callback as a positional argument if it is last.

v2.3.1 (2014-12-07)

- Add `pipe` and `pipe_right` as aliases of `flow` and `flow_right`.
- Fix `merge` so that trailing `{}` or `[]` don't overwrite previous source values.
- Make `py_` an alias for `_`.

v2.3.0 (2014-11-10)

- Support type callbacks (e.g. `int`, `float`, `str`, etc.) by only passing a single callback argument when invoking the callback.
- Drop official support for Python 3.2. Too many testing dependencies no longer work on it.

v2.2.0 (2014-10-28)

- Add `append`.
- Add `deep_get`.
- Add `deep_has`.
- Add `deep_map_values`.
- Add `deep_set`.
- Add `deep_pluck`.
- Add `deep_property`.
- Add `join`.
- Add `pop`.
- Add `push`.
- Add `reverse`.
- Add `shift`.
- Add `sort`.

- Add `splice`.
- Add `unshift`.
- Add `url`.
- Fix bug in `snake_case` that resulted in returned string not being converted to lower case.
- Fix bug in chaining method access test which skipped the actual test.
- Make `_` instance alias method access to methods with a trailing underscore in their name. For example, `_.map()` becomes an alias for `map_()`.
- Make `deep_prop` an alias of `deep_property`.
- Make `has` work with deep paths.
- Make `has_path` an alias of `deep_has`.
- Make `get_path` handle escaping the `.` delimiter for string keys.
- Make `get_path` handle list indexing using strings such as `'0.1.2'` to access `'value'` in `[[0, [0, 0, 'value']]]`.
- Make `concat` an alias of `cat`.

v2.1.0 (2014-09-17)

- Add `add`, `sum_`.
- Add `average`, `avg`, `mean`.
- Add `mapiter`.
- Add `median`.
- Add `moving_average`, `moving_avg`.
- Add `power`, `pow_`.
- Add `round_`, `curve`.
- Add `scale`.
- Add `slope`.
- Add `std_deviation`, `sigma`.
- Add `transpose`.
- Add `variance`.
- Add `zscore`.

v2.0.0 (2014-09-11)

- Add `_` instance that supports both method chaining and module method calling.
- Add `cat`.
- Add `conjoin`.
- Add `deburr`.
- Add `disjoin`.

- Add explode.
- Add flatten_deep.
- Add flow.
- Add flow_right.
- Add get_path.
- Add has_path.
- Add implode.
- Add intercalate.
- Add interleave.
- Add intersperse.
- Add is_associative.
- Add is_even.
- Add is_float.
- Add is_decreasing.
- Add is_increasing.
- Add is_indexed.
- Add is_instance_of.
- Add is_integer.
- Add is_json.
- Add is_monotone.
- Add is_negative.
- Add is_odd.
- Add is_positive.
- Add is_strictly_decreasing.
- Add is_strictly_increasing.
- Add is_zero.
- Add iterated.
- Add js_match.
- Add js_replace.
- Add juxtapose.
- Add mapcat.
- Add reductions.
- Add reductions_right.
- Add rename_keys.
- Add set_path.
- Add split_at.

- Add `thru`.
- Add `to_string`.
- Add `update_path`.
- Add `words`.
- Make callback function calling adapt to argspec of given callback function. If, for example, the full callback signature is `(item, index, obj)` but the passed in callback only supports `(item)`, then only `item` will be passed in when callback is invoked. Previously, callbacks had to support all arguments or implement star-args.
- Make `chain` lazy and only compute the final value when `value` called.
- Make `compose` an alias of `flow_right`.
- Make `flatten` shallow by default, remove callback option, and add `is_deep` option. **(breaking change)**
- Make `is_number` return `False` for boolean `True` and `False`. **(breaking change)**
- Make `invert` accept multivalue argument.
- Make `result` accept default argument.
- Make `slice_` accept optional start and end arguments.
- Move files in `pydash/api/` to `pydash/`. **(breaking change)**
- Move predicate functions from `pydash.api.objects` to `pydash.api.predicates`. **(breaking change)**
- Rename `create_callback` to `iteratee`. **(breaking change)**
- Rename functions to callables in order to allow `functions.py` to exist at the root of the pydash module folder. **(breaking change)**
- Rename *private* utility function `_iter_callback` to `itercallback`. **(breaking change)**
- Rename *private* utility function `_iter_list_callback` to `iterlist_callback`. **(breaking change)**
- Rename *private* utility function `_iter_dict_callback` to `iterdict_callback`. **(breaking change)**
- Rename *private* utility function `_iterate` to `iterator`. **(breaking change)**
- Rename *private* utility function `_iter_dict` to `iterdict`. **(breaking change)**
- Rename *private* utility function `_iter_list` to `iterlist`. **(breaking change)**
- Rename *private* utility function `_iter_unique` to `iterunique`. **(breaking change)**
- Rename *private* utility function `_get_item` to `getitem`. **(breaking change)**
- Rename *private* utility function `_set_item` to `setitem`. **(breaking change)**
- Rename *private* utility function `_deprecated` to `deprecated`. **(breaking change)**
- Undeprecate `tail` and make alias of `rest`.

v1.1.0 (2014-08-19)

- Add `attempt`.
- Add `before`.
- Add `camel_case`.
- Add `capitalize`.

- Add `chunk`.
- Add `curry_right`.
- Add `drop_right`.
- Add `drop_right_while`.
- Add `drop_while`.
- Add `ends_with`.
- Add `escape_reg_exp` and `escape_re`.
- Add `is_error`.
- Add `is_reg_exp` and `is_re`.
- Add `kebab_case`.
- Add `keys_in` as alias of `keys`.
- Add `negate`.
- Add `pad`.
- Add `pad_left`.
- Add `pad_right`.
- Add `partition`.
- Add `pull_at`.
- Add `repeat`.
- Add `slice_`.
- Add `snake_case`.
- Add `sorted_last_index`.
- Add `starts_with`.
- Add `take_right`.
- Add `take_right_while`.
- Add `take_while`.
- Add `trim`.
- Add `trim_left`.
- Add `trim_right`.
- Add `trunc`.
- Add `values_in` as alias of `values`.
- Create `pydash.api.strings` module.
- Deprecate `tail`.
- Modify `drop` to accept `n` argument and remove as alias of `rest`.
- Modify `take` to accept `n` argument and remove as alias of `first`.
- Move `escape` and `unescape` from `pydash.api.utilities` to `pydash.api.strings`. (**breaking change**)

- Move `range_` from `pydash.api.arrays` to `pydash.api.utilities`. (breaking change)

v1.0.0 (2014-08-05)

- Add Python 2.6 and Python 3 support.
- Add `after`.
- Add `assign` and `extend`. Thanks [nathancahill](#)!
- Add `callback` and `create_callback`.
- Add `chain`.
- Add `clone`.
- Add `clone_deep`.
- Add `compose`.
- Add `constant`.
- Add `count_by`. Thanks [nathancahill](#)!
- Add `curry`.
- Add `debounce`.
- Add `defaults`. Thanks [nathancahill](#)!
- Add `delay`.
- Add `escape`.
- Add `find_key`. Thanks [nathancahill](#)!
- Add `find_last`. Thanks [nathancahill](#)!
- Add `find_last_index`. Thanks [nathancahill](#)!
- Add `find_last_key`. Thanks [nathancahill](#)!
- Add `for_each`. Thanks [nathancahill](#)!
- Add `for_each_right`. Thanks [nathancahill](#)!
- Add `for_in`. Thanks [nathancahill](#)!
- Add `for_in_right`. Thanks [nathancahill](#)!
- Add `for_own`. Thanks [nathancahill](#)!
- Add `for_own_right`. Thanks [nathancahill](#)!
- Add `functions_` and `methods`. Thanks [nathancahill](#)!
- Add `group_by`. Thanks [nathancahill](#)!
- Add `has`. Thanks [nathancahill](#)!
- Add `index_by`. Thanks [nathancahill](#)!
- Add `identity`.
- Add `inject`.
- Add `invert`.
- Add `invoke`. Thanks [nathancahill](#)!

- Add `is_list`. Thanks [nathancahill](#)!
- Add `is_boolean`. Thanks [nathancahill](#)!
- Add `is_empty`. Thanks [nathancahill](#)!
- Add `is_equal`.
- Add `is_function`. Thanks [nathancahill](#)!
- Add `is_none`. Thanks [nathancahill](#)!
- Add `is_number`. Thanks [nathancahill](#)!
- Add `is_object`.
- Add `is_plain_object`.
- Add `is_string`. Thanks [nathancahill](#)!
- Add `keys`.
- Add `map_values`.
- Add `matches`.
- Add `max_`. Thanks [nathancahill](#)!
- Add `memoize`.
- Add `merge`.
- Add `min_`. Thanks [nathancahill](#)!
- Add `noop`.
- Add `now`.
- Add `omit`.
- Add `once`.
- Add `pairs`.
- Add `parse_int`.
- Add `partial`.
- Add `partial_right`.
- Add `pick`.
- Add `property_ and prop`.
- Add `pull`. Thanks [nathancahill](#)!
- Add `random`.
- Add `reduce_ and foldl`.
- Add `reduce_right and foldr`.
- Add `reject`. Thanks [nathancahill](#)!
- Add `remove`.
- Add `result`.
- Add `sample`.
- Add `shuffle`.

- Add `size`.
- Add `sort_by`. Thanks [nathancahill](#)!
- Add `tap`.
- Add `throttle`.
- Add `times`.
- Add `transform`.
- Add `to_list`. Thanks [nathancahill](#)!
- Add `unescape`.
- Add `unique_id`.
- Add `values`.
- Add `wrap`.
- Add `xor`.

v0.0.0 (2014-07-22)

- Add `all_`.
- Add `any_`.
- Add `at`.
- Add `bisect_left`.
- Add `collect`.
- Add `collections`.
- Add `compact`.
- Add `contains`.
- Add `detect`.
- Add `difference`.
- Add `drop`.
- Add `each`.
- Add `each_right`.
- Add `every`.
- Add `filter_`.
- Add `find`.
- Add `find_index`.
- Add `find_where`.
- Add `first`.
- Add `flatten`.
- Add `head`.
- Add `include`.

- Add `index_of`.
- Add `initial`.
- Add `intersection`.
- Add `last`.
- Add `last_index_of`.
- Add `map_`.
- Add `object_`.
- Add `pluck`.
- Add `range_`.
- Add `rest`.
- Add `select`.
- Add `some`.
- Add `sorted_index`.
- Add `tail`.
- Add `take`.
- Add `union`.
- Add `uniq`.
- Add `unique`.
- Add `unzip`.
- Add `where`.
- Add `without`.
- Add `zip_`.
- Add `zip_object`.

Authors

Lead

- Derrick Gilland, dgilland@gmail.com, [dgilland@github](https://github.com/dgilland)

Contributors

- Nathan Cahill, nathan@nathancahill.com, [nathancahill@github](https://github.com/nathancahill)
- Klaus Sevensleeper, k7sleeper@gmail.com, [k7sleeper@github](https://github.com/k7sleeper)
- Bharadwaj Yarlagadda, yarlagaddabharadwaj@gmail.com, [bharadwajyarlagadda@github](https://github.com/bharadwajyarlagadda)
- Michael James, [urbnjames](https://github.com/urbnjames)
- Tim Griesser, tgriesser@gmail.com, [tgriesser@github](https://github.com/tgriesser)

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at <https://github.com/dgilland/pydash>.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” or “help wanted” is open to whoever wants to implement it.

Write Documentation

Pydash could always use more documentation, whether as part of the official pydash docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at <https://github.com/dgilland/pydash>.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here’s how to set up pydash for local development.

1. Fork the pydash repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pydash.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenv installed, this is how you set up your fork for local development:

```
$ cd pydash
$ pip install -r requirements-dev.txt
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass linting and all unit tests by testing with tox across all supported Python versions:

```
$ invoke tox
```

6. Add yourself to `AUTHORS.rst`.
7. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the `README.rst`.
3. The pull request should work for Python 2.7, 3.4, and 3.5. Check https://travis-ci.org/dgilland/pydash/pull_requests and make sure that the tests pass for all supported Python versions.

Kudos

Thank you to [Lodash](#) for providing such a great library to port.

CHAPTER 6

Indices and Tables

- `genindex`
- `modindex`
- `search`

p

- `pydash.arrays`, [22](#)
- `pydash.chaining`, [52](#)
- `pydash.collections`, [53](#)
- `pydash.functions`, [67](#)
- `pydash.numerical`, [76](#)
- `pydash.objects`, [86](#)
- `pydash.predicates`, [106](#)
- `pydash.strings`, [123](#)
- `pydash.utilities`, [148](#)

A

`add()` (in module `pydash.numerical`), 76
`after()` (in module `pydash.functions`), 67
`ary()` (in module `pydash.functions`), 67
`assign()` (in module `pydash.objects`), 86
`assign_with()` (in module `pydash.objects`), 87
`at()` (in module `pydash.collections`), 53
`attempt()` (in module `pydash.utilities`), 148

B

`before()` (in module `pydash.functions`), 68

C

`callables()` (in module `pydash.objects`), 87
`camel_case()` (in module `pydash.strings`), 123
`capitalize()` (in module `pydash.strings`), 123
`ceil()` (in module `pydash.numerical`), 77
`chain()` (in module `pydash.chaining`), 52
`chars()` (in module `pydash.strings`), 125
`chop()` (in module `pydash.strings`), 124
`chop_right()` (in module `pydash.strings`), 124
`chunk()` (in module `pydash.arrays`), 22
`clamp()` (in module `pydash.numerical`), 77
`clean()` (in module `pydash.strings`), 125
`clone()` (in module `pydash.objects`), 88
`clone_deep()` (in module `pydash.objects`), 88
`clone_deep_with()` (in module `pydash.objects`), 89
`clone_with()` (in module `pydash.objects`), 89
`compact()` (in module `pydash.arrays`), 23
`concat()` (in module `pydash.arrays`), 23
`cond()` (in module `pydash.utilities`), 148
`conforms()` (in module `pydash.utilities`), 148
`conforms_to()` (in module `pydash.utilities`), 149
`conjoin()` (in module `pydash.functions`), 68
`constant()` (in module `pydash.utilities`), 149
`count_by()` (in module `pydash.collections`), 54
`count_substr()` (in module `pydash.strings`), 125
`curry()` (in module `pydash.functions`), 69
`curry_right()` (in module `pydash.functions`), 69

D

`debounce()` (in module `pydash.functions`), 70
`deburr()` (in module `pydash.strings`), 126
`decapitalize()` (in module `pydash.strings`), 126
`default_to()` (in module `pydash.utilities`), 150
`defaults()` (in module `pydash.objects`), 89
`defaults_deep()` (in module `pydash.objects`), 90
`delay()` (in module `pydash.functions`), 70
`difference()` (in module `pydash.arrays`), 23
`difference_by()` (in module `pydash.arrays`), 24
`difference_with()` (in module `pydash.arrays`), 24
`disjoin()` (in module `pydash.functions`), 70
`divide()` (in module `pydash.numerical`), 78
`drop()` (in module `pydash.arrays`), 24
`drop_right()` (in module `pydash.arrays`), 25
`drop_right_while()` (in module `pydash.arrays`), 25
`drop_while()` (in module `pydash.arrays`), 26
`duplicates()` (in module `pydash.arrays`), 26

E

`ends_with()` (in module `pydash.strings`), 126
`ensure_ends_with()` (in module `pydash.strings`), 127
`ensure_starts_with()` (in module `pydash.strings`), 127
`eq()` (in module `pydash.predicates`), 106
`escape()` (in module `pydash.strings`), 127
`escape_reg_exp()` (in module `pydash.strings`), 128
`every()` (in module `pydash.collections`), 54

F

`fill()` (in module `pydash.arrays`), 26
`filter_()` (in module `pydash.collections`), 55
`find()` (in module `pydash.collections`), 55
`find_index()` (in module `pydash.arrays`), 27
`find_key()` (in module `pydash.objects`), 90
`find_last()` (in module `pydash.collections`), 56
`find_last_index()` (in module `pydash.arrays`), 27
`find_last_key()` (in module `pydash.objects`), 91
`flat_map()` (in module `pydash.collections`), 56
`flat_map_deep()` (in module `pydash.collections`), 56

`flat_map_depth()` (in module `pydash.collections`), 57
`flatten()` (in module `pydash.arrays`), 28
`flatten_deep()` (in module `pydash.arrays`), 28
`flatten_depth()` (in module `pydash.arrays`), 28
`flip()` (in module `pydash.functions`), 71
`floor()` (in module `pydash.numerical`), 78
`flow()` (in module `pydash.functions`), 71
`flow_right()` (in module `pydash.functions`), 72
`for_each()` (in module `pydash.collections`), 57
`for_each_right()` (in module `pydash.collections`), 57
`for_in()` (in module `pydash.objects`), 91
`for_in_right()` (in module `pydash.objects`), 91
`from_pairs()` (in module `pydash.arrays`), 29

G

`get()` (in module `pydash.objects`), 92
`group_by()` (in module `pydash.collections`), 58
`gt()` (in module `pydash.predicates`), 106
`gte()` (in module `pydash.predicates`), 107

H

`has()` (in module `pydash.objects`), 92
`has_substr()` (in module `pydash.strings`), 128
`head()` (in module `pydash.arrays`), 29
`human_case()` (in module `pydash.strings`), 128

I

`identity()` (in module `pydash.utilities`), 150
`in_range()` (in module `pydash.predicates`), 108
`includes()` (in module `pydash.collections`), 58
`index_of()` (in module `pydash.arrays`), 29
`initial()` (in module `pydash.arrays`), 30
`insert_substr()` (in module `pydash.strings`), 129
`intercalate()` (in module `pydash.arrays`), 30
`interleave()` (in module `pydash.arrays`), 30
`intersection()` (in module `pydash.arrays`), 31
`intersection_by()` (in module `pydash.arrays`), 31
`intersection_with()` (in module `pydash.arrays`), 31
`interperse()` (in module `pydash.arrays`), 32
`invert()` (in module `pydash.objects`), 93
`invert_by()` (in module `pydash.objects`), 94
`invoke()` (in module `pydash.objects`), 94
`invoke_map()` (in module `pydash.collections`), 59
`is_associative()` (in module `pydash.predicates`), 109
`is_blank()` (in module `pydash.predicates`), 109
`is_boolean()` (in module `pydash.predicates`), 109
`is_builtin()` (in module `pydash.predicates`), 110
`is_date()` (in module `pydash.predicates`), 110
`is_decreasing()` (in module `pydash.predicates`), 111
`is_dict()` (in module `pydash.predicates`), 111
`is_empty()` (in module `pydash.predicates`), 111
`is_equal()` (in module `pydash.predicates`), 112
`is_equal_with()` (in module `pydash.predicates`), 112
`is_error()` (in module `pydash.predicates`), 113

`is_even()` (in module `pydash.predicates`), 113
`is_float()` (in module `pydash.predicates`), 113
`is_function()` (in module `pydash.predicates`), 114
`is_increasing()` (in module `pydash.predicates`), 114
`is_indexed()` (in module `pydash.predicates`), 114
`is_instance_of()` (in module `pydash.predicates`), 115
`is_integer()` (in module `pydash.predicates`), 115
`is_iterable()` (in module `pydash.predicates`), 116
`is_json()` (in module `pydash.predicates`), 116
`is_list()` (in module `pydash.predicates`), 116
`is_match()` (in module `pydash.predicates`), 117
`is_match_with()` (in module `pydash.predicates`), 117
`is_monotone()` (in module `pydash.predicates`), 118
`is_nan()` (in module `pydash.predicates`), 118
`is_negative()` (in module `pydash.predicates`), 119
`is_none()` (in module `pydash.predicates`), 119
`is_number()` (in module `pydash.predicates`), 119
`is_object()` (in module `pydash.predicates`), 120
`is_odd()` (in module `pydash.predicates`), 120
`is_positive()` (in module `pydash.predicates`), 120
`is_reg_exp()` (in module `pydash.predicates`), 121
`is_set()` (in module `pydash.predicates`), 121
`is_strictly_decreasing()` (in module `pydash.predicates`), 121
`is_strictly_increasing()` (in module `pydash.predicates`), 122
`is_string()` (in module `pydash.predicates`), 122
`is_tuple()` (in module `pydash.predicates`), 122
`is_zero()` (in module `pydash.predicates`), 123
`iterated()` (in module `pydash.functions`), 72
`iteratee()` (in module `pydash.utilities`), 150

J

`join()` (in module `pydash.strings`), 129
`juxtapose()` (in module `pydash.functions`), 72

K

`kebab_case()` (in module `pydash.strings`), 129
`key_by()` (in module `pydash.collections`), 59
`keys()` (in module `pydash.objects`), 95

L

`last()` (in module `pydash.arrays`), 32
`last_index_of()` (in module `pydash.arrays`), 33
`lines()` (in module `pydash.strings`), 130
`lower_case()` (in module `pydash.strings`), 130
`lower_first()` (in module `pydash.strings`), 130
`lt()` (in module `pydash.predicates`), 107
`lte()` (in module `pydash.predicates`), 108

M

`map_()` (in module `pydash.collections`), 60
`map_keys()` (in module `pydash.objects`), 95

[map_values\(\)](#) (in module `pydash.objects`), 95
[map_values_deep\(\)](#) (in module `pydash.objects`), 96
[mapcat\(\)](#) (in module `pydash.arrays`), 33
[matches\(\)](#) (in module `pydash.utilities`), 151
[matches_property\(\)](#) (in module `pydash.utilities`), 152
[max_\(\)](#) (in module `pydash.numerical`), 79
[max_by\(\)](#) (in module `pydash.numerical`), 79
[mean\(\)](#) (in module `pydash.numerical`), 79
[mean_by\(\)](#) (in module `pydash.numerical`), 80
[median\(\)](#) (in module `pydash.numerical`), 80
[memoize\(\)](#) (in module `pydash.utilities`), 152
[merge\(\)](#) (in module `pydash.objects`), 96
[merge_with\(\)](#) (in module `pydash.objects`), 97
[method\(\)](#) (in module `pydash.utilities`), 153
[method_of\(\)](#) (in module `pydash.utilities`), 153
[min_\(\)](#) (in module `pydash.numerical`), 80
[min_by\(\)](#) (in module `pydash.numerical`), 81
[moving_mean\(\)](#) (in module `pydash.numerical`), 81
[multiply\(\)](#) (in module `pydash.numerical`), 82

N

[negate\(\)](#) (in module `pydash.functions`), 73
[noop\(\)](#) (in module `pydash.utilities`), 154
[now\(\)](#) (in module `pydash.utilities`), 154
[nth\(\)](#) (in module `pydash.arrays`), 33
[nth_arg\(\)](#) (in module `pydash.utilities`), 154
[number_format\(\)](#) (in module `pydash.strings`), 131

O

[omit\(\)](#) (in module `pydash.objects`), 97
[omit_by\(\)](#) (in module `pydash.objects`), 98
[once\(\)](#) (in module `pydash.functions`), 73
[order_by\(\)](#) (in module `pydash.collections`), 60
[over\(\)](#) (in module `pydash.utilities`), 154
[over_args\(\)](#) (in module `pydash.functions`), 73
[over_every\(\)](#) (in module `pydash.utilities`), 155
[over_some\(\)](#) (in module `pydash.utilities`), 155

P

[pad\(\)](#) (in module `pydash.strings`), 131
[pad_end\(\)](#) (in module `pydash.strings`), 132
[pad_start\(\)](#) (in module `pydash.strings`), 132
[parse_int\(\)](#) (in module `pydash.objects`), 98
[partial\(\)](#) (in module `pydash.functions`), 74
[partial_right\(\)](#) (in module `pydash.functions`), 74
[partition\(\)](#) (in module `pydash.collections`), 61
[pascal_case\(\)](#) (in module `pydash.strings`), 133
[pick\(\)](#) (in module `pydash.objects`), 99
[pick_by\(\)](#) (in module `pydash.objects`), 99
[pluck\(\)](#) (in module `pydash.collections`), 61
[power\(\)](#) (in module `pydash.numerical`), 82
[predecessor\(\)](#) (in module `pydash.strings`), 133
[property_\(\)](#) (in module `pydash.utilities`), 155
[property_of\(\)](#) (in module `pydash.utilities`), 156

[prune\(\)](#) (in module `pydash.strings`), 133
[pull\(\)](#) (in module `pydash.arrays`), 34
[pull_all\(\)](#) (in module `pydash.arrays`), 34
[pull_all_by\(\)](#) (in module `pydash.arrays`), 35
[pull_all_with\(\)](#) (in module `pydash.arrays`), 35
[pull_at\(\)](#) (in module `pydash.arrays`), 36
[push\(\)](#) (in module `pydash.arrays`), 36
[pydash.arrays](#) (module), 22
[pydash.chaining](#) (module), 52
[pydash.collections](#) (module), 53
[pydash.functions](#) (module), 67
[pydash.numerical](#) (module), 76
[pydash.objects](#) (module), 86
[pydash.predicates](#) (module), 106
[pydash.strings](#) (module), 123
[pydash.utilities](#) (module), 148

Q

[quote\(\)](#) (in module `pydash.strings`), 134

R

[random\(\)](#) (in module `pydash.utilities`), 156
[range_\(\)](#) (in module `pydash.utilities`), 156
[range_right\(\)](#) (in module `pydash.utilities`), 157
[rearg\(\)](#) (in module `pydash.functions`), 75
[reduce_\(\)](#) (in module `pydash.collections`), 62
[reduce_right\(\)](#) (in module `pydash.collections`), 62
[reductions\(\)](#) (in module `pydash.collections`), 63
[reductions_right\(\)](#) (in module `pydash.collections`), 63
[reg_exp_js_match\(\)](#) (in module `pydash.strings`), 134
[reg_exp_js_replace\(\)](#) (in module `pydash.strings`), 135
[reg_exp_replace\(\)](#) (in module `pydash.strings`), 135
[reject\(\)](#) (in module `pydash.collections`), 64
[remove\(\)](#) (in module `pydash.arrays`), 36
[rename_keys\(\)](#) (in module `pydash.objects`), 99
[repeat\(\)](#) (in module `pydash.strings`), 136
[replace\(\)](#) (in module `pydash.strings`), 136
[result\(\)](#) (in module `pydash.utilities`), 157
[reverse\(\)](#) (in module `pydash.arrays`), 37
[round_\(\)](#) (in module `pydash.numerical`), 83

S

[sample\(\)](#) (in module `pydash.collections`), 64
[sample_size\(\)](#) (in module `pydash.collections`), 64
[scale\(\)](#) (in module `pydash.numerical`), 83
[separator_case\(\)](#) (in module `pydash.strings`), 137
[series_phrase\(\)](#) (in module `pydash.strings`), 137
[series_phrase_serial\(\)](#) (in module `pydash.strings`), 138
[set_\(\)](#) (in module `pydash.objects`), 100
[set_with\(\)](#) (in module `pydash.objects`), 100
[shift\(\)](#) (in module `pydash.arrays`), 37
[shuffle\(\)](#) (in module `pydash.collections`), 65
[size\(\)](#) (in module `pydash.collections`), 65

`slice_()` (in module `pydash.arrays`), 38
`slope()` (in module `pydash.numerical`), 83
`slugify()` (in module `pydash.strings`), 138
`snake_case()` (in module `pydash.strings`), 139
`some()` (in module `pydash.collections`), 65
`sort()` (in module `pydash.arrays`), 38
`sort_by()` (in module `pydash.collections`), 66
`sorted_index()` (in module `pydash.arrays`), 39
`sorted_index_by()` (in module `pydash.arrays`), 39
`sorted_index_of()` (in module `pydash.arrays`), 40
`sorted_last_index()` (in module `pydash.arrays`), 40
`sorted_last_index_by()` (in module `pydash.arrays`), 41
`sorted_last_index_of()` (in module `pydash.arrays`), 41
`sorted_uniq()` (in module `pydash.arrays`), 41
`sorted_uniq_by()` (in module `pydash.arrays`), 42
`splice()` (in module `pydash.arrays`), 42
`split()` (in module `pydash.strings`), 139
`split_at()` (in module `pydash.arrays`), 43
`spread()` (in module `pydash.functions`), 75
`start_case()` (in module `pydash.strings`), 139
`starts_with()` (in module `pydash.strings`), 140
`std_deviation()` (in module `pydash.numerical`), 84
`strip_tags()` (in module `pydash.strings`), 140
`stub_dict()` (in module `pydash.utilities`), 158
`stub_false()` (in module `pydash.utilities`), 158
`stub_list()` (in module `pydash.utilities`), 158
`stub_string()` (in module `pydash.utilities`), 159
`stub_true()` (in module `pydash.utilities`), 159
`substr_left()` (in module `pydash.strings`), 140
`substr_left_end()` (in module `pydash.strings`), 141
`substr_right()` (in module `pydash.strings`), 141
`substr_right_end()` (in module `pydash.strings`), 141
`subtract()` (in module `pydash.numerical`), 85
`successor()` (in module `pydash.strings`), 142
`sum_()` (in module `pydash.numerical`), 84
`sum_by()` (in module `pydash.numerical`), 84
`surround()` (in module `pydash.strings`), 142
`swap_case()` (in module `pydash.strings`), 143

T

`tail()` (in module `pydash.arrays`), 43
`take()` (in module `pydash.arrays`), 44
`take_right()` (in module `pydash.arrays`), 44
`take_right_while()` (in module `pydash.arrays`), 44
`take_while()` (in module `pydash.arrays`), 45
`tap()` (in module `pydash.chaining`), 52
`throttle()` (in module `pydash.functions`), 75
`thru()` (in module `pydash.chaining`), 53
`times()` (in module `pydash.utilities`), 159
`title_case()` (in module `pydash.strings`), 143
`to_boolean()` (in module `pydash.objects`), 101
`to_dict()` (in module `pydash.objects`), 101
`to_integer()` (in module `pydash.objects`), 102
`to_list()` (in module `pydash.collections`), 66

`to_lower()` (in module `pydash.strings`), 143
`to_number()` (in module `pydash.objects`), 102
`to_pairs()` (in module `pydash.objects`), 103
`to_path()` (in module `pydash.utilities`), 160
`to_string()` (in module `pydash.objects`), 103
`to_upper()` (in module `pydash.strings`), 144
`transform()` (in module `pydash.objects`), 103
`transpose()` (in module `pydash.numerical`), 85
`trim()` (in module `pydash.strings`), 144
`trim_end()` (in module `pydash.strings`), 144
`trim_start()` (in module `pydash.strings`), 145
`truncate()` (in module `pydash.strings`), 145

U

`unary()` (in module `pydash.functions`), 76
`unescape()` (in module `pydash.strings`), 145
`union()` (in module `pydash.arrays`), 45
`union_by()` (in module `pydash.arrays`), 45
`union_with()` (in module `pydash.arrays`), 46
`uniq()` (in module `pydash.arrays`), 46
`uniq_by()` (in module `pydash.arrays`), 47
`uniq_with()` (in module `pydash.arrays`), 47
`unique_id()` (in module `pydash.utilities`), 160
`unquote()` (in module `pydash.strings`), 146
`unset()` (in module `pydash.objects`), 104
`unshift()` (in module `pydash.arrays`), 47
`unzip()` (in module `pydash.arrays`), 48
`unzip_with()` (in module `pydash.arrays`), 48
`update()` (in module `pydash.objects`), 104
`update_with()` (in module `pydash.objects`), 105
`upper_case()` (in module `pydash.strings`), 146
`upper_first()` (in module `pydash.strings`), 147
`url()` (in module `pydash.strings`), 147

V

`values()` (in module `pydash.objects`), 105
`variance()` (in module `pydash.numerical`), 85

W

`without()` (in module `pydash.arrays`), 48
`words()` (in module `pydash.strings`), 147
`wrap()` (in module `pydash.functions`), 76

X

`xor()` (in module `pydash.arrays`), 49
`xor_by()` (in module `pydash.arrays`), 49
`xor_with()` (in module `pydash.arrays`), 50

Z

`zip_()` (in module `pydash.arrays`), 50
`zip_object()` (in module `pydash.arrays`), 50
`zip_object_deep()` (in module `pydash.arrays`), 51
`zip_with()` (in module `pydash.arrays`), 51
`zscore()` (in module `pydash.numerical`), 86