pydash Documentation

Release 7.0.3

Derrick Gilland

May 04, 2023

CONTENTS

The kitchen sink of Python utility libraries for doing "stuff" in a functional way. Based on the Lo-Dash Javascript library.

CHAPTER

ONE

NOTE

Looking for a library that is more memory efficient and better suited for large datasets? Check out fnc! It's built around generators and iteration and has iteratee-first function signatures.

CHAPTER

TWO

LINKS

- Project: https://github.com/dgilland/pydash
- Documentation: http://pydash.readthedocs.org
- PyPi: https://pypi.python.org/pypi/pydash/
- Github Actions: https://github.com/dgilland/pydash/actions

CHAPTER

THREE

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

# Arrays
>>> pydash.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]]]
```

(continues on next page)

(continued from previous page)

```
# 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*.

CHAPTER

FOUR

GUIDE

4.1 Installation

pydash requires Python >= 3.6. It has no external dependencies.

To install from PyPi:

```
pip install pydash
```

4.2 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

# Arrays
>>> pydash.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)
```

(continues on next page)

(continued from previous page)

```
[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*.

4.3 Lodash Differences

4.3.1 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 `callables()`. 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.

4.3.2 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 evaulate to `[1, 2]` and not `[3, 4]` (as would be the case for `'a.b'`).

4.3.3 Extra Functions

In addition to porting Lodash, pydash contains functions found in lodashcontrib, lodashdeep, lodashmath, and underscorestring.

4.3.4 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.

4.3.5 Templating

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

4.4 Callbacks

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

4.4.1 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}
... ]
```

(continues on next page)

(continued from previous page)

```
# 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
```

4.4.2 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
```

4.4.3 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']
```

4.4.4 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
```

4.4.5 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]
```

4.5 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.

4.6 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()
```

4.6.1 Lazy Evaluation

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

```
>>> 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
```

4.6.2 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())
```

4.6.3 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
```

4.6.4 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
```

4.6.5 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
```

4.7 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.

4.8 Upgrading

4.8.1 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()*

4.8.2 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 differnt 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*.

4.8.3 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 `callables()` 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*.

4.9 Developer Guide

This guide provides an overview of the tooling this project uses and how to execute developer workflows using the developer CLI.

4.9.1 Python Environments

This Python project is tested against different Python versions. For local development, it is a good idea to have those versions installed so that tests can be run against each.

There are libraries that can help with this. Which tools to use is largely a matter of preference, but below are a few recommendations.

For managing multiple Python versions:

- pyenv
- OS package manager (e.g. apt, yum, homebrew, etc)
- Build from source

For managing Python virtualenvs:

- pyenv-virtualenv
- pew
- python-venv

4.9.2 Tooling

The following tools are used by this project:

Tool	Description	Configuration
black	Code formatter	`pyproject.toml`
isort	Import statement formatter	`setup.cfg`
docformatter	Docstring formatter	`setup.cfg`
flake8	Code linter	`setup.cfg`
pylint	Code linter	`pylintrc`
mypy	Type checker	`setup.cfg`
pytest	Test framework	`setup.cfg`
tox	Test environment manager	`tox.ini`
invoke	CLI task execution library	`tasks.py`

4.9.3 Workflows

The following workflows use developer CLI commands via invoke and are defined in `tasks.py`.

Autoformat Code

To run all autoformatters:

```
inv fmt
```

This is the same as running each autoformatter individually:

```
inv black
inv isort
inv docformatter
```

Lint

To run all linters:

```
inv lint
```

This is the same as running each linter individually:

```
inv flake8
inv pylint
inv mypy
```

Test

To run all unit tests:

```
inv unit
```

To run unit tests and builds:

```
inv test
```

Test on All Supported Python Versions

To run tests on all supported Python versions:

```
tox
```

This requires that the supported versions are available on the PATH.

Build Package

To build the package:

```
inv build
```

This will output the source and binary distributions under `dist/`.

Build Docs

To build documentation:

```
inv docs
```

This will output the documentation under `docs/_build/`.

Serve Docs

To serve docs over HTTP:

```
inv docs -s|--server [-b|--bind 127.0.0.1] [-p|--port 8000]

inv docs -s
inv docs -s -p 8080
inv docs -s -b 0.0.0.0 -p 8080
```

Delete Build Files

To remove all build and temporary files:

```
inv clean
```

This will remove Python bytecode files, egg files, build output folders, caches, and tox folders.

Release Package

To release a new version of the package to https://pypi.org:

```
inv release
```

4.9.4 CI/CD

This project uses Github Actions for CI/CD:

- https://github.com/dgilland/pydash/actions

CHAPTER

FIVE

API REFERENCE

Includes links to source code.

5.1 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.

5.1.1 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]
```

(continues on next page)

(continued from previous page)

```
# 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_()*

5.1.2 Arrays

Functions that operate on lists.

New in version 1.0.0.

pydash.arrays.**chunk**(*array: Sequence[pydash.arrays.T]*, *size: int = 1*) → List[Sequence[pydash.arrays.T]]

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 to chunk.
- **size** – Chunk size. Defaults to 1.

Returns New list containing chunks of *array*.

Example

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

New in version 1.1.0.

pydash.arrays.**compact**(*array: Iterable[Optional[pydash.arrays.T]]*) → List[pydash.arrays.T]
Creates a list with all falsey values of array removed.

Parameters **array** – List to compact.

Returns Compacted list.

Example

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

New in version 1.0.0.

pydash.arrays.**concat**(**arrays: Iterable[pydash.arrays.T]*) → List[pydash.arrays.T]
Concatenates zero or more lists into one.

Parameters **arrays** – Lists to concatenate.

Returns Concatenated 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: Iterable[pydash.arrays.T]*, **others: Iterable[pydash.arrays.T]*) → List[pydash.arrays.T]
Creates a list of list elements not present in others.

Parameters

- **array** – List to process.
- **others** – Lists to check.

Returns Difference between *others*.

Example

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

New in version 1.0.0.

pydash.arrays.**difference_by**(*array: Iterable[pydash.arrays.T]*, **others: Iterable[pydash.arrays.T]*, *iteratee: Optional[Union[int, str, List, Tuple, Dict, Callable[[pydash.arrays.T], Any]]]*) → List[pydash.arrays.T]

pydash.arrays.**difference_by**(*array: Iterable[pydash.arrays.T]*, **others: Union[int, str, List, Tuple, Dict, Iterable[pydash.arrays.T], Callable[[pydash.arrays.T], Any]]*) → List[pydash.arrays.T]

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

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

Returns Difference between *others*.

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: Iterable[pydash.arrays.T]*, **others: Iterable[pydash.arrays.T2]*, *comparator: Optional[Callable[[pydash.arrays.T, pydash.arrays.T2], Any]]*) → List[pydash.arrays.T]

pydash.arrays.**difference_with**(*array: Iterable[pydash.arrays.T]*, **others: Union[Iterable[pydash.arrays.T2], Callable[[pydash.arrays.T, pydash.arrays.T2], Any]]*) → List[pydash.arrays.T]

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

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

Returns Difference between *others*.

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: Sequence[pydash.arrays.T]*, *n: int = 1*) → List[pydash.arrays.T]

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

Parameters

- **array** – List to process.
- **n** – Number of elements to drop. Defaults to 1.

Returns Dropped 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: Sequence[pydash.arrays.T]*, *n: int = 1*) → List[pydash.arrays.T]

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

Parameters

- **array** – List to process.
- **n** – Number of elements to drop. Defaults to 1.

Returns Dropped 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: Sequence[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T, int, List[pydash.arrays.T]], Any]*) → List[pydash.arrays.T]

pydash.arrays.**drop_right_while**(*array: Sequence[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T, int], Any]*) → List[pydash.arrays.T]

pydash.arrays.**drop_right_while**(*array: Sequence[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T], Any]*) → List[pydash.arrays.T]

pydash.arrays.**drop_right_while**(*array: Sequence[pydash.arrays.T]*, *predicate: None = None*) → List[pydash.arrays.T]

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 to process.
- **predicate** – Predicate called per iteration

Returns Dropped 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: Sequence[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T, int, List[pydash.arrays.T]], Any]*) → List[pydash.arrays.T]
pydash.arrays.**drop_while**(*array: Sequence[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T, int], Any]*) → List[pydash.arrays.T]
pydash.arrays.**drop_while**(*array: Sequence[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T], Any]*) → List[pydash.arrays.T]
pydash.arrays.**drop_while**(*array: Sequence[pydash.arrays.T]*, *predicate: None = None*) → List[pydash.arrays.T]

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 to process.
- **predicate** – Predicate called per iteration

Returns Dropped list.

Example

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

New in version 1.1.0.

pydash.arrays.**duplicates**(*array: Sequence[pydash.arrays.T]*, *iteratee: Optional[Union[Callable[[pydash.arrays.T], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.arrays.T]

Creates a unique list of duplicate values from *array*. If iteratee is passed, each element of array is passed through an 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 to process.

- **iteratee** – Iteratee applied per iteration.

Returns List of duplicates.

Example

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

New in version 3.0.0.

pydash.arrays.**fill**(*array: Sequence[pydash.arrays.T]*, *value: pydash.arrays.T2*, *start: int = 0*, *end: Optional[int] = None*) → List[Union[pydash.arrays.T, pydash.arrays.T2]]

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

Parameters

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

Returns Filled *array*.

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: Iterable[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T, int, List[pydash.arrays.T]], Any]*) → int

pydash.arrays.**find_index**(*array: Iterable[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T, int], Any]*) → int

pydash.arrays.**find_index**(*array: Iterable[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T], Any]*) → int

pydash.arrays.**find_index**(*array: Iterable[Any]*, *predicate: None = None*) → int

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 to process.
- **predicate** – Predicate applied per iteration.

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

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: Iterable[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T, int, List[pydash.arrays.T]], Any]*) → int
pydash.arrays.**find_last_index**(*array: Iterable[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T, int], Any]*) → int
pydash.arrays.**find_last_index**(*array: Iterable[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T], Any]*) → int
pydash.arrays.**find_last_index**(*array: Iterable[Any]*, *predicate: None = None*) → int

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

Parameters

- **array** – List to process.
- **predicate** – Predicate applied per iteration.

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

Example

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

New in version 1.0.0.

pydash.arrays.**flatten**(*array: Iterable[Iterable[pydash.arrays.T]]*) → List[pydash.arrays.T]
pydash.arrays.**flatten**(*array: Iterable[pydash.arrays.T]*) → List[pydash.arrays.T]

Flattens array a single level deep.

Parameters **array** – List to flatten.

Returns Flattened 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: Iterable*) → List

Flattens an array recursively.

Parameters **array** – List to flatten.

Returns Flattened list.

Example

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

New in version 2.0.0.

pydash.arrays.**flatten_depth**(*array: Iterable*, *depth: int = 1*) → List
Recursively flatten *array* up to *depth* times.

Parameters

- **array** – List to flatten.
- **depth** – Depth to flatten to. Defaults to 1.

Returns Flattened 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: Iterable[Tuple[pydash.arrays.T, pydash.arrays.T2]]*) → Dict[pydash.arrays.T, pydash.arrays.T2]
pydash.arrays.**from_pairs**(*pairs: Iterable[List[Union[pydash.arrays.T, pydash.arrays.T2]]]*) → Dict[Union[pydash.arrays.T, pydash.arrays.T2], Union[pydash.arrays.T, pydash.arrays.T2]]
Returns a dict from the given list of pairs.

Parameters **pairs** – List of key-value pairs.

Returns dict

Example

```
>>> from_pairs([['a', 1], ['b', 2]]) == {'a': 1, 'b': 2}
True
```

New in version 4.0.0.

pydash.arrays.**head**(*array: Sequence[pydash.arrays.T]*) → Optional[pydash.arrays.T]
Return the first element of *array*.

Parameters **array** – List to process.

Returns First element of list.

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: Sequence[pydash.arrays.T]*, *value: pydash.arrays.T*, *from_index: int = 0*) → int

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

Parameters

- **array** – List to search.
- **value** – Value to search for.
- **from_index** – Index to search from.

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

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: Sequence[pydash.arrays.T]*) → Sequence[pydash.arrays.T]

Return all but the last element of *array*.

Parameters **array** – List to process.

Returns Initial part of *array*.

Example

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

New in version 1.0.0.

pydash.arrays.**intercalate**(*array: Iterable[Iterable[pydash.arrays.T]]*, *separator: pydash.arrays.T2*) → List[Union[pydash.arrays.T, pydash.arrays.T2]]

pydash.arrays.**intercalate**(*array: Iterable[pydash.arrays.T]*, *separator: pydash.arrays.T2*) → List[Union[pydash.arrays.T, pydash.arrays.T2]]

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

Parameters

- **array** – List to intercalate.
- **separator** – Element to insert.

Returns Intercalated 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: Iterable[pydash.arrays.T]*) → List[pydash.arrays.T]

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** – Lists to interleave.

Returns 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: Sequence[pydash.arrays.T]*, **others: Iterable[Any]*) → List[pydash.arrays.T]

Computes the intersection of all the passed-in arrays.

Parameters

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

Returns Intersection of provided lists.

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: Sequence[pydash.arrays.T]*, **others: Iterable[Any]*, *iteratee: Union[Callable[[pydash.arrays.T], Any], int, str, List, Tuple, Dict]*) → List[pydash.arrays.T]

pydash.arrays.**intersection_by**(*array: Sequence[pydash.arrays.T]*, **others: Union[Iterable[Any], Callable[[pydash.arrays.T], Any], int, str, List, Tuple, Dict]*) → List[pydash.arrays.T]

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

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

Returns Intersection of provided lists.

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: Sequence[pydash.arrays.T], *others: Iterable[pydash.arrays.T2], comparator: Callable[[pydash.arrays.T, pydash.arrays.T2], Any]*) → List[pydash.arrays.T]

pydash.arrays.**intersection_with**(*array: Sequence[pydash.arrays.T], *others: Union[Iterable[pydash.arrays.T2], Callable[[pydash.arrays.T, pydash.arrays.T2], Any]]*) → List[pydash.arrays.T]

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

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

Returns Intersection of provided lists.

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: Iterable[pydash.arrays.T], separator: pydash.arrays.T2*) → List[Union[pydash.arrays.T, pydash.arrays.T2]]

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

Parameters

- **array** – List to intersperse.
- **separator** – Element to insert.

Returns Interspersed 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: Sequence[pydash.arrays.T]*) → Optional[pydash.arrays.T]
Return the last element of *array*.

Parameters **array** – List to process.

Returns Last part of *array*.

Example

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

New in version 1.0.0.

pydash.arrays.**last_index_of**(*array: Sequence[Any]*, *value: Any*, *from_index: Optional[int] = None*) → int
Gets the index at which the last occurrence of value is found.

Parameters

- **array** – List to search.
- **value** – Value to search for.
- **from_index** – Index to search from.

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

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: Iterable[pydash.arrays.T]*, *iteratee: Callable[[pydash.arrays.T, int, List[pydash.arrays.T]], Union[List[pydash.arrays.T2], List[List[pydash.arrays.T2]]]]*) → List[pydash.arrays.T2]
pydash.arrays.**mapcat**(*array: Iterable[pydash.arrays.T]*, *iteratee: Callable[[pydash.arrays.T, int, List[pydash.arrays.T]], pydash.arrays.T2]*) → List[pydash.arrays.T2]
pydash.arrays.**mapcat**(*array: Iterable[pydash.arrays.T]*, *iteratee: Callable[[pydash.arrays.T, int], Union[List[pydash.arrays.T2], List[List[pydash.arrays.T2]]]]*) → List[pydash.arrays.T2]
pydash.arrays.**mapcat**(*array: Iterable[pydash.arrays.T]*, *iteratee: Callable[[pydash.arrays.T, int], pydash.arrays.T2]*) → List[pydash.arrays.T2]
pydash.arrays.**mapcat**(*array: Iterable[pydash.arrays.T]*, *iteratee: Callable[[pydash.arrays.T], Union[List[pydash.arrays.T2], List[List[pydash.arrays.T2]]]]*) → List[pydash.arrays.T2]
pydash.arrays.**mapcat**(*array: Iterable[pydash.arrays.T]*, *iteratee: Callable[[pydash.arrays.T], pydash.arrays.T2]*) → List[pydash.arrays.T2]

pydash.arrays.**mapcat**(*array: Iterable[Union[List[pydash.arrays.T], List[List[pydash.arrays.T]]]]*, *iteratee: None = None*) → List[Union[pydash.arrays.T, List[pydash.arrays.T]]]

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

Parameters

- **array** – List to map and concatenate.
- **iteratee** – Iteratee to apply to each element.

Returns Mapped and concatenated 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: Iterable[pydash.arrays.T]*, *pos: int = 0*) → Optional[pydash.arrays.T]

Gets the element at index n of array.

Parameters

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

Returns Returns the element at `pos`.

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: List[pydash.arrays.T]*, **values: pydash.arrays.T*) → List[pydash.arrays.T]

Removes all provided values from the given array.

Parameters

- **array** – List to pull from.
- **values** – Values to remove.

Returns Modified *array*.

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: List[pydash.arrays.T]*, *values: Iterable[pydash.arrays.T]*) → List[pydash.arrays.T]

Removes all provided values from the given array.

Parameters

- **array** – Array to modify.
- **values** – Values to remove.

Returns Modified *array*.

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: List[pydash.arrays.T]*, *values: Iterable[pydash.arrays.T]*, *iteratee: Optional[Union[Callable[[pydash.arrays.T], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.arrays.T]

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** – Array to modify.
- **values** – Values to remove.
- **iteratee** – Function to transform the elements of the arrays. Defaults to *identity()*.

Returns Modified *array*.

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: List[pydash.arrays.T]*, *values: Iterable[pydash.arrays.T]*, *comparator: Optional[Callable[[pydash.arrays.T, pydash.arrays.T], Any]] = None*) → List[pydash.arrays.T]

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** – Array to modify.
- **values** – Values to remove.
- **comparator** – Function to compare the elements of the arrays. Defaults to *is_equal()*.

Returns Modified *array*.

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: List[pydash.arrays.T]*, **indexes: int*) → List[pydash.arrays.T]

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 to pull from.
- **indexes** – Indexes to pull.

Returns Modified *array*.

> **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: List[pydash.arrays.T]*, **items: pydash.arrays.T2*) → List[Union[pydash.arrays.T, pydash.arrays.T2]]

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

Parameters

- **array** – List to push to.
- **items** – Items to append.

Returns Modified *array*.

> **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: List[pydash.arrays.T]*, *predicate: Optional[Union[Callable[[pydash.arrays.T, int, List[pydash.arrays.T]], Any], Callable[[pydash.arrays.T, int], Any], Callable[[pydash.arrays.T], Any]]] = None*) → List[pydash.arrays.T]

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

Parameters

- **array** – List to remove elements from.
- **predicate** – Predicate applied per iteration.

Returns Removed elements of *array*.

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: pydash.arrays.SequenceT*) → pydash.arrays.SequenceT

Return *array* in reverse order.

Parameters **array** – Object to process.

Returns Reverse of object.

Example

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

New in version 2.2.0.

pydash.arrays.**shift**(*array: List[pydash.arrays.T]*) → pydash.arrays.T

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

Parameters **array** – List to shift.

Returns First element of *array*.

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: pydash.arrays.SequenceT*, *start: int = 0*, *end: Optional[int] = None*) → pydash.arrays.SequenceT

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

Parameters

- **array** – Array to slice.
- **start** – Start index. Defaults to `0`.
- **end** – End index. Defaults to selecting the value at `start` index.

Returns Sliced 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: List[SupportsRichComparisonT]*, *comparator: None = None*, *key: None = None*, *reverse: bool = False*) → List[SupportsRichComparisonT]
pydash.arrays.**sort**(*array: List[pydash.arrays.T]*, *comparator: Callable[[pydash.arrays.T, pydash.arrays.T], int]*, ***, *reverse: bool = 'False'*) → List[pydash.arrays.T]
pydash.arrays.**sort**(*array: List[pydash.arrays.T]*, ***, *key: Callable[[pydash.arrays.T], SupportsRichComparisonT]*, *reverse: bool = 'False'*) → List[pydash.arrays.T]

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 to sort.

- **comparator** – 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** – A function of one argument used to extract a comparator key from each list element. Defaults to `None`. This argument is mutually exclusive with *comparator*.
- **reverse** – Whether to reverse the sort. Defaults to `False`.

Returns Sorted 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: Sequence[SupportsRichComparisonT]*, *value: SupportsRichComparisonT*) → int

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 to inspect.
- **value** – Value to evaluate.

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

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: Sequence[pydash.arrays.T]*, *value: pydash.arrays.T*, *iteratee: Union[int, str, List, Tuple, Dict, Callable[[pydash.arrays.T], SupportsRichComparisonT]]*) → int

pydash.arrays.**sorted_index_by**(*array: Sequence[SupportsRichComparisonT]*, *value: SupportsRichComparisonT*, *iteratee: None = None*) → int

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 to inspect.
- **value** – Value to evaluate.
- **iteratee** – The iteratee invoked per element. Defaults to *identity()*.

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

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: Sequence[SupportsRichComparisonT]*, *value: SupportsRichComparisonT*) → int

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

Parameters

- **array** – Array to inspect.
- **value** – Value to search for.

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

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: Sequence[SupportsRichComparisonT]*, *value: SupportsRichComparisonT*) → int

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 to inspect.
- **value** – Value to evaluate.

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

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: Sequence[pydash.arrays.T]*, *value: pydash.arrays.T*, *iteratee: Union[int, str, List, Tuple, Dict, Callable[[pydash.arrays.T], SupportsRichComparisonT]]*) → int

pydash.arrays.**sorted_last_index_by**(*array: Sequence[SupportsRichComparisonT]*, *value: SupportsRichComparisonT*, *iteratee: None = None*) → int

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 to inspect.
- **value** – Value to evaluate.
- **iteratee** – The iteratee invoked per element. Defaults to *identity()*.

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

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: Sequence[SupportsRichComparisonT]*, *value: SupportsRichComparisonT*) → int

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

Parameters

- **array** – Array to inspect.
- **value** – Value to search for.

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

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: Iterable[SupportsRichComparisonT]*) → List[SupportsRichComparisonT]

Return sorted array with unique elements.

Parameters **array** – List of values to be sorted.

Returns List of unique elements in a sorted fashion.

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: Iterable[SupportsRichComparisonT]*, *iteratee: Optional[Callable[[SupportsRichComparisonT], SupportsRichComparisonT]] = None*) → List[SupportsRichComparisonT]

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 of values to be sorted.
- **iteratee** – Function to transform the elements of the arrays. Defaults to *identity()*.

Returns Unique 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: pydash.arrays.MutableSequenceT*, *start: int*, *count: Optional[int] = None*, **items: Any*) → pydash.arrays.MutableSequenceT

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

Parameters

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

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

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: Sequence[pydash.arrays.T]*, *index: int*) → List[Sequence[pydash.arrays.T]]

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

Parameters

- **array** – List to split.
- **index** – Index to split at.

Returns Split list.

Example

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

New in version 2.0.0.

pydash.arrays.**tail**(*array: Sequence[pydash.arrays.T]*) → Sequence[pydash.arrays.T]

Return all but the first element of *array*.

Parameters **array** – List to process.

Returns Rest of the 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: Sequence[pydash.arrays.T]*, *n: int = 1*) → Sequence[pydash.arrays.T]
Creates a slice of *array* with *n* elements taken from the beginning.

Parameters

- **array** – List to process.
- **n** – Number of elements to take. Defaults to 1.

Returns Taken 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: Sequence[pydash.arrays.T]*, *n: int = 1*) → Sequence[pydash.arrays.T]
Creates a slice of *array* with *n* elements taken from the end.

Parameters

- **array** – List to process.
- **n** – Number of elements to take. Defaults to 1.

Returns Taken 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: Sequence[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T, int, List[pydash.arrays.T]], Any]*) → Sequence[pydash.arrays.T]
pydash.arrays.**take_right_while**(*array: Sequence[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T, int], Any]*) → Sequence[pydash.arrays.T]
pydash.arrays.**take_right_while**(*array: Sequence[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T], Any]*) → Sequence[pydash.arrays.T]

pydash.arrays.**take_right_while**(*array: Sequence[pydash.arrays.T]*, *predicate: None = None*) → Sequence[pydash.arrays.T]

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 to process.
- **predicate** – Predicate called per iteration

Returns Dropped 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: Sequence[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T, int, List[pydash.arrays.T]], Any]*) → List[pydash.arrays.T]
pydash.arrays.**take_while**(*array: Sequence[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T, int], Any]*) → List[pydash.arrays.T]
pydash.arrays.**take_while**(*array: Sequence[pydash.arrays.T]*, *predicate: Callable[[pydash.arrays.T], Any]*) → List[pydash.arrays.T]
pydash.arrays.**take_while**(*array: Sequence[pydash.arrays.T]*, *predicate: None = None*) → List[pydash.arrays.T]

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 to process.
- **predicate** – Predicate called per iteration

Returns Taken list.

Example

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

New in version 1.1.0.

pydash.arrays.**union**(*array: Sequence[pydash.arrays.T]*) → List[pydash.arrays.T]
pydash.arrays.**union**(*array: Sequence[pydash.arrays.T]*, **others: Sequence[pydash.arrays.T2]*) → List[Union[pydash.arrays.T, pydash.arrays.T2]]

Computes the union of the passed-in arrays.

Parameters

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

Returns Unionized 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: Sequence[pydash.arrays.T]*, **others: Iterable[pydash.arrays.T]*, *iteratee: Callable[[pydash.arrays.T], Any]*) → List[pydash.arrays.T]
pydash.arrays.**union_by**(*array: Sequence[pydash.arrays.T]*, **others: Union[Iterable[pydash.arrays.T], Callable[[pydash.arrays.T], Any]]*) → List[pydash.arrays.T]

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

Parameters

- **array** – List to unionize with.
- **others** – Lists to unionize with *array*.

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

Returns Unionized 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: Sequence[pydash.arrays.T]*, **others: Iterable[pydash.arrays.T2]*, *comparator: Callable[[pydash.arrays.T, pydash.arrays.T2], Any]*) → List[pydash.arrays.T]
pydash.arrays.**union_with**(*array: Sequence[pydash.arrays.T]*, **others: Union[Iterable[pydash.arrays.T2], Callable[[pydash.arrays.T, pydash.arrays.T2], Any]]*) → List[pydash.arrays.T]

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 to unionize with.
- **others** – Lists to unionize with *array*.

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

Returns Unionized 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: Iterable[pydash.arrays.T]*) → List[pydash.arrays.T]

Creates a duplicate-value-free version of the array. If iteratee is passed, each element of array is passed through an 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 to process.

Returns Unique 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: Iterable[pydash.arrays.T]*, *iteratee: Optional[Callable[[pydash.arrays.T], Any]] = None*) → List[pydash.arrays.T]

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 to process.
- **iteratee** – Function to transform the elements of the arrays. Defaults to *identity()*.

Returns Unique 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: Sequence[pydash.arrays.T]*, *comparator: Optional[Callable[[pydash.arrays.T, pydash.arrays.T], Any]] = None*) → List[pydash.arrays.T]

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 to process.
- **comparator** – Function to compare the elements of the arrays. Defaults to *is_equal()*.

Returns Unique 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: List[pydash.arrays.T]*, **items: pydash.arrays.T2*) → List[Union[pydash.arrays.T, pydash.arrays.T2]]

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

Parameters

- **array** – List to modify.
- **items** – Items to insert.

Returns Modified 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: Iterable[Iterable[pydash.arrays.T]]*) → List[List[pydash.arrays.T]]

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 to process.

Returns Unzipped 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: Iterable[Iterable[pydash.arrays.T]]*, *iteratee: Union[Callable[[pydash.arrays.T, pydash.arrays.T, int, List[pydash.arrays.T]], pydash.arrays.T2], Callable[[pydash.arrays.T, pydash.arrays.T, int], pydash.arrays.T2], Callable[[pydash.arrays.T, pydash.arrays.T], pydash.arrays.T2], Callable[[pydash.arrays.T], pydash.arrays.T2]]*) → List[pydash.arrays.T2]
pydash.arrays.**unzip_with**(*array: Iterable[Iterable[pydash.arrays.T]]*, *iteratee: None = None*) → List[List[pydash.arrays.T]]

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

Parameters

- **array** – List to process.
- **iteratee** – Function to combine regrouped values.

Returns Unzipped 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: Iterable[pydash.arrays.T]*, **values: pydash.arrays.T*) → List[pydash.arrays.T]

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

Parameters

- **array** – List to filter.
- **values** – Values to remove.

Returns Filtered list.

Example

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

New in version 1.0.0.

pydash.arrays.**xor**(*array: Iterable[pydash.arrays.T]*, **lists: Iterable[pydash.arrays.T]*) → List[pydash.arrays.T]

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

Parameters

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

Returns XOR'd list.

Example

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

New in version 1.0.0.

pydash.arrays.**xor_by**(*array: Iterable[pydash.arrays.T]*, **lists: Iterable[pydash.arrays.T]*, *iteratee: Union[Callable[[pydash.arrays.T], Any], int, str, List, Tuple, Dict]*) → List[pydash.arrays.T]

pydash.arrays.**xor_by**(*array: Iterable[pydash.arrays.T]*, **lists: Union[Iterable[pydash.arrays.T], Callable[[pydash.arrays.T], Any]]*) → List[pydash.arrays.T]

This method is like *xor()* except that it accepts iteratee which is invoked for each element of each arras 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 to process.
- ***lists** – Lists to xor with.

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

Returns XOR'd 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: Sequence[pydash.arrays.T]*, **lists: Iterable[pydash.arrays.T2]*, *comparator: Callable[[pydash.arrays.T, pydash.arrays.T2], Any]*) → List[pydash.arrays.T]

pydash.arrays.**xor_with**(*array: Sequence[pydash.arrays.T], *lists: Union[Iterable[pydash.arrays.T2], Callable[[pydash.arrays.T, pydash.arrays.T2], Any]]*) → List[pydash.arrays.T]

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 to process.
- ***lists** – Lists to xor with.

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

Returns XOR'd 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: Iterable[pydash.arrays.T]*) → List[List[pydash.arrays.T]]

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

Parameters **arrays** – Lists to process.

Returns Zipped 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: Iterable[Tuple[pydash.arrays.T, pydash.arrays.T2]], values: None = None*) → Dict[pydash.arrays.T, pydash.arrays.T2]

pydash.arrays.**zip_object**(*keys: Iterable[List[Union[pydash.arrays.T, pydash.arrays.T2]]], values: None = None*) → Dict[Union[pydash.arrays.T, pydash.arrays.T2], Union[pydash.arrays.T, pydash.arrays.T2]]

pydash.arrays.**zip_object**(*keys: Iterable[pydash.arrays.T], values: List[pydash.arrays.T2]*) → Dict[pydash.arrays.T, pydash.arrays.T2]

Creates a dict composed of 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** – Either a list of keys or a list of `[key, value]` pairs.
- **values** – List of values to zip.

Returns Zipped 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: Iterable[Any]*, *values: Optional[List[Any]] = None*) → Dict

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

Parameters

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

Returns Zipped 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: Iterable[pydash.arrays.T]*, *iteratee: Union[Callable[[pydash.arrays.T, pydash.arrays.T, int, List[pydash.arrays.T]], pydash.arrays.T2], Callable[[pydash.arrays.T, pydash.arrays.T, int], pydash.arrays.T2], Callable[[pydash.arrays.T, pydash.arrays.T], pydash.arrays.T2], Callable[[pydash.arrays.T], pydash.arrays.T2]]*) → List[pydash.arrays.T2]
pydash.arrays.**zip_with**(**arrays: Iterable[pydash.arrays.T]*) → List[List[pydash.arrays.T]]
pydash.arrays.**zip_with**(**arrays: Union[Iterable[pydash.arrays.T], Callable[[pydash.arrays.T, pydash.arrays.T, int, List[pydash.arrays.T]], pydash.arrays.T2], Callable[[pydash.arrays.T, pydash.arrays.T, int], pydash.arrays.T2], Callable[[pydash.arrays.T, pydash.arrays.T], pydash.arrays.T2], Callable[[pydash.arrays.T], pydash.arrays.T2]]*) → List[Union[List[pydash.arrays.T], pydash.arrays.T2]]

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** – Lists to process.

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

Returns Zipped list of grouped elements.

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.

5.1.3 Chaining

pydash.chaining.**chain**(*value: Union[pydash.chaining.chaining.T, pydash.helpers.Unset] = <pydash.helpers.Unset object>*) → pydash.chaining.chaining.Chain[pydash.chaining.chaining.T]

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** – Value to initialize chain operations with.

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

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: pydash.chaining.chaining.T*, *interceptor: Callable[[pydash.chaining.chaining.T], Any]*) → pydash.chaining.chaining.T

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** – Current value of chain operation.
- **interceptor** – Function called on *value*.

Returns *value* after *interceptor* call.

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: pydash.chaining.chaining.T*, *interceptor: Callable[[pydash.chaining.chaining.T], pydash.chaining.chaining.T2]*) → pydash.chaining.chaining.T2

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** – Current value of chain operation.
- **interceptor** – Function called with *value*.

Returns Results of `interceptor(value)`.

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.

5.1.4 Collections

Functions that operate on lists and dicts.

New in version 1.0.0.

pydash.collections.**at**(*collection: Mapping[pydash.collections.T, pydash.collections.T2]*, **paths: pydash.collections.T*) → List[Optional[pydash.collections.T2]]
pydash.collections.**at**(*collection: Mapping[pydash.collections.T, Any]*, **paths: Union[pydash.collections.T, Iterable[pydash.collections.T]]*) → List[Any]
pydash.collections.**at**(*collection: Iterable[pydash.collections.T]*, **paths: int*) → List[Optional[pydash.collections.T]]
pydash.collections.**at**(*collection: Iterable[Any]*, **paths: Union[int, Iterable[int]]*) → List[Any]

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** – Collection to iterate over.
- ***paths** – The indexes of *collection* to retrieve, specified as individual indexes or arrays of indexes.

Returns filtered list

Example

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

New in version 1.0.0.

Changed in version 4.1.0: Support deep path access.

pydash.collections.**count_by**(*collection: Mapping[Any, pydash.collections.T2]*, *iteratee: None = None*) → Dict[pydash.collections.T2, int]
pydash.collections.**count_by**(*collection: Mapping[pydash.collections.T, pydash.collections.T2]*, *iteratee: Callable[[pydash.collections.T2, pydash.collections.T, Dict[pydash.collections.T, pydash.collections.T2]], pydash.collections.T3]*) → Dict[pydash.collections.T3, int]
pydash.collections.**count_by**(*collection: Mapping[pydash.collections.T, pydash.collections.T2]*, *iteratee: Callable[[pydash.collections.T2, pydash.collections.T], pydash.collections.T3]*) → Dict[pydash.collections.T3, int]

pydash.collections.**count_by**(*collection: Mapping[Any, pydash.collections.T2]*, *iteratee: Callable[[pydash.collections.T2], pydash.collections.T3]*) → Dict[pydash.collections.T3, int]
pydash.collections.**count_by**(*collection: Iterable[pydash.collections.T]*, *iteratee: None = None*) → Dict[pydash.collections.T, int]
pydash.collections.**count_by**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T, int, List[pydash.collections.T]], pydash.collections.T2]*) → Dict[pydash.collections.T2, int]
pydash.collections.**count_by**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T, int], pydash.collections.T2]*) → Dict[pydash.collections.T2, int]
pydash.collections.**count_by**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T], pydash.collections.T2]*) → Dict[pydash.collections.T2, int]

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

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.

Returns Dict containing counts by key.

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: Iterable[pydash.collections.T]*, *predicate: Optional[Union[Callable[[pydash.collections.T], Any], int, str, List, Tuple, Dict]] = None*) → bool

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 *matches()* style predicate will return `True` for elements that have the properties of the given object, else `False`.

Parameters

- **collection** – Collection to iterate over.
- **predicate** – Predicate applied per iteration.

Returns Whether all elements are truthy.

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: Mapping[pydash.collections.T, pydash.collections.T2]*, *predicate: Optional[Union[Callable[[pydash.collections.T2, pydash.collections.T, Dict[pydash.collections.T, pydash.collections.T2]], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T2]
pydash.collections.**filter_**(*collection: Mapping[pydash.collections.T, pydash.collections.T2]*, *predicate: Optional[Union[Callable[[pydash.collections.T2, pydash.collections.T], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T2]
pydash.collections.**filter_**(*collection: Mapping[Any, pydash.collections.T2]*, *predicate: Optional[Union[Callable[[pydash.collections.T2], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T2]
pydash.collections.**filter_**(*collection: Iterable[pydash.collections.T]*, *predicate: Optional[Union[Callable[[pydash.collections.T, int, List[pydash.collections.T]], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T]
pydash.collections.**filter_**(*collection: Iterable[pydash.collections.T]*, *predicate: Optional[Union[Callable[[pydash.collections.T, int], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T]
pydash.collections.**filter_**(*collection: Iterable[pydash.collections.T]*, *predicate: Optional[Union[Callable[[pydash.collections.T], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T]

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

Parameters

- **collection** – Collection to iterate over.
- **predicate** – Predicate applied per iteration.

Returns Filtered 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: Dict[pydash.collections.T, pydash.collections.T2], predicate: Optional[Union[Callable[[pydash.collections.T2, pydash.collections.T, Dict[pydash.collections.T, pydash.collections.T2]], Any], int, str, List, Tuple, Dict]] = None*) → Optional[pydash.collections.T2]
pydash.collections.**find**(*collection: Dict[pydash.collections.T, pydash.collections.T2], predicate: Optional[Union[Callable[[pydash.collections.T2, pydash.collections.T], Any], int, str, List, Tuple, Dict]] = None*) → Optional[pydash.collections.T2]
pydash.collections.**find**(*collection: Dict[pydash.collections.T, pydash.collections.T2], predicate: Optional[Union[Callable[[pydash.collections.T2], Any], int, str, List, Tuple, Dict]] = None*) → Optional[pydash.collections.T2]
pydash.collections.**find**(*collection: List[pydash.collections.T], predicate: Optional[Union[Callable[[pydash.collections.T, int, List[pydash.collections.T]], Any], int, str, List, Tuple, Dict]] = None*) → Optional[pydash.collections.T]
pydash.collections.**find**(*collection: List[pydash.collections.T], predicate: Optional[Union[Callable[[pydash.collections.T, int], Any], int, str, List, Tuple, Dict]] = None*) → Optional[pydash.collections.T]
pydash.collections.**find**(*collection: List[pydash.collections.T], predicate: Optional[Union[Callable[[pydash.collections.T], Any], int, str, List, Tuple, Dict]] = None*) → Optional[pydash.collections.T]

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

Parameters

- **collection** – Collection to iterate over.
- **predicate** – Predicate applied per iteration.

Returns First element found or `None`.

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: Dict[pydash.collections.T, pydash.collections.T2], predicate: Optional[Union[Callable[[pydash.collections.T2, pydash.collections.T, Dict[pydash.collections.T, pydash.collections.T2]], Any], int, str, List, Tuple, Dict]] = None*) → Optional[pydash.collections.T2]

pydash.collections.**find_last**(*collection: Dict[pydash.collections.T, pydash.collections.T2], predicate: Optional[Union[Callable[[pydash.collections.T2, pydash.collections.T], Any], int, str, List, Tuple, Dict]] = None*) → Optional[pydash.collections.T2]
pydash.collections.**find_last**(*collection: Dict[Any, pydash.collections.T2], predicate: Optional[Union[Callable[[pydash.collections.T2], Any], int, str, List, Tuple, Dict]] = None*) → Optional[pydash.collections.T2]
pydash.collections.**find_last**(*collection: List[pydash.collections.T], predicate: Optional[Union[Callable[[pydash.collections.T, int, List[pydash.collections.T]], Any], int, str, List, Tuple, Dict]] = None*) → Optional[pydash.collections.T]
pydash.collections.**find_last**(*collection: List[pydash.collections.T], predicate: Optional[Union[Callable[[pydash.collections.T, int], Any], int, str, List, Tuple, Dict]] = None*) → Optional[pydash.collections.T]
pydash.collections.**find_last**(*collection: List[pydash.collections.T], predicate: Optional[Union[Callable[[pydash.collections.T], Any], int, str, List, Tuple, Dict]] = None*) → Optional[pydash.collections.T]

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

Parameters

- **collection** – Collection to iterate over.
- **predicate** – Predicate applied per iteration.

Returns Last element found or `None`.

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: Mapping[pydash.collections.T, pydash.collections.T2], iteratee: Callable[[pydash.collections.T2, pydash.collections.T, Dict[pydash.collections.T, pydash.collections.T2]], Iterable[pydash.collections.T3]]*) → List[pydash.collections.T3]
pydash.collections.**flat_map**(*collection: Mapping[pydash.collections.T, pydash.collections.T2], iteratee: Callable[[pydash.collections.T2, pydash.collections.T], Iterable[pydash.collections.T3]]*) → List[pydash.collections.T3]
pydash.collections.**flat_map**(*collection: Mapping[Any, pydash.collections.T2], iteratee: Callable[[pydash.collections.T2], Iterable[pydash.collections.T3]]*) → List[pydash.collections.T3]
pydash.collections.**flat_map**(*collection: Mapping[pydash.collections.T, pydash.collections.T2], iteratee: Callable[[pydash.collections.T2, pydash.collections.T, Dict[pydash.collections.T, pydash.collections.T2]], pydash.collections.T3]*) → List[pydash.collections.T3]
pydash.collections.**flat_map**(*collection: Mapping[pydash.collections.T, pydash.collections.T2], iteratee: Callable[[pydash.collections.T2, pydash.collections.T], pydash.collections.T3]*) → List[pydash.collections.T3]
pydash.collections.**flat_map**(*collection: Mapping[Any, pydash.collections.T2], iteratee: Callable[[pydash.collections.T2], pydash.collections.T3]*) → List[pydash.collections.T3]

pydash.collections.**flat_map**(*collection: Mapping[Any, Iterable[pydash.collections.T2]]*, *iteratee: None = None*) → List[pydash.collections.T2]
pydash.collections.**flat_map**(*collection: Mapping[Any, pydash.collections.T2]*, *iteratee: None = None*) → List[pydash.collections.T2]
pydash.collections.**flat_map**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T, int, List[pydash.collections.T]], Iterable[pydash.collections.T2]]*) → List[pydash.collections.T2]
pydash.collections.**flat_map**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T, int], Iterable[pydash.collections.T2]]*) → List[pydash.collections.T2]
pydash.collections.**flat_map**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T], Iterable[pydash.collections.T2]]*) → List[pydash.collections.T2]
pydash.collections.**flat_map**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T, int, List[pydash.collections.T]], pydash.collections.T2]*) → List[pydash.collections.T2]
pydash.collections.**flat_map**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T, int], pydash.collections.T2]*) → List[pydash.collections.T2]
pydash.collections.**flat_map**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T], pydash.collections.T2]*) → List[pydash.collections.T2]
pydash.collections.**flat_map**(*collection: Iterable[Iterable[pydash.collections.T]]*, *iteratee: None = None*) → List[pydash.collections.T]
pydash.collections.**flat_map**(*collection: Iterable[pydash.collections.T]*, *iteratee: None = None*) → List[pydash.collections.T]

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

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.

Returns Flattened mapped 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: Mapping[pydash.collections.T, pydash.collections.T2]*, *iteratee: Optional[Callable[[pydash.collections.T2, pydash.collections.T, Dict[pydash.collections.T, pydash.collections.T2]], Any]] = None*) → List
pydash.collections.**flat_map_deep**(*collection: Mapping[pydash.collections.T, pydash.collections.T2]*, *iteratee: Optional[Callable[[pydash.collections.T2, pydash.collections.T], Any]] = None*) → List
pydash.collections.**flat_map_deep**(*collection: Mapping[Any, pydash.collections.T2]*, *iteratee: Optional[Callable[[pydash.collections.T2], Any]] = None*) → List

pydash.collections.**flat_map_deep**(*collection: Iterable[pydash.collections.T], iteratee: Optional[Callable[[pydash.collections.T, int, List[pydash.collections.T]], Any]] = None*) → List
pydash.collections.**flat_map_deep**(*collection: Iterable[pydash.collections.T], iteratee: Optional[Callable[[pydash.collections.T, int], Any]] = None*) → List
pydash.collections.**flat_map_deep**(*collection: Iterable[pydash.collections.T], iteratee: Optional[Callable[[pydash.collections.T], Any]] = None*) → List

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

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.

Returns Flattened mapped 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: Mapping[pydash.collections.T, pydash.collections.T2], iteratee: Optional[Callable[[pydash.collections.T2, pydash.collections.T, Dict[pydash.collections.T, pydash.collections.T2]], Any]] = None, depth: int = 1*) → List
pydash.collections.**flat_map_depth**(*collection: Mapping[pydash.collections.T, pydash.collections.T2], iteratee: Optional[Callable[[pydash.collections.T2, pydash.collections.T], Any]] = None, depth: int = 1*) → List
pydash.collections.**flat_map_depth**(*collection: Mapping[Any, pydash.collections.T2], iteratee: Optional[Callable[[pydash.collections.T2], Any]] = None, depth: int = 1*) → List
pydash.collections.**flat_map_depth**(*collection: Iterable[pydash.collections.T], iteratee: Optional[Callable[[pydash.collections.T, int, List[pydash.collections.T]], Any]] = None, depth: int = 1*) → List
pydash.collections.**flat_map_depth**(*collection: Iterable[pydash.collections.T], iteratee: Optional[Callable[[pydash.collections.T, int], Any]] = None, depth: int = 1*) → List
pydash.collections.**flat_map_depth**(*collection: Iterable[pydash.collections.T], iteratee: Optional[Callable[[pydash.collections.T], Any]] = None, depth: int = 1*) → List

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

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.

Returns Flattened mapped 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: Dict[pydash.collections.T, pydash.collections.T2]*, *iteratee: Optional[Union[Callable[[pydash.collections.T2, pydash.collections.T, Dict[pydash.collections.T, pydash.collections.T2]], Any], int, str, List, Tuple, Dict]] = None*) → Dict[pydash.collections.T, pydash.collections.T2]
pydash.collections.**for_each**(*collection: Dict[pydash.collections.T, pydash.collections.T2]*, *iteratee: Optional[Union[Callable[[pydash.collections.T2, pydash.collections.T], Any], int, str, List, Tuple, Dict]] = None*) → Dict[pydash.collections.T, pydash.collections.T2]
pydash.collections.**for_each**(*collection: Dict[pydash.collections.T, pydash.collections.T2]*, *iteratee: Optional[Union[Callable[[pydash.collections.T2], Any], int, str, List, Tuple, Dict]] = None*) → Dict[pydash.collections.T, pydash.collections.T2]
pydash.collections.**for_each**(*collection: List[pydash.collections.T]*, *iteratee: Optional[Union[Callable[[pydash.collections.T, int, List[pydash.collections.T]], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T]
pydash.collections.**for_each**(*collection: List[pydash.collections.T]*, *iteratee: Optional[Union[Callable[[pydash.collections.T, int], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T]
pydash.collections.**for_each**(*collection: List[pydash.collections.T]*, *iteratee: Optional[Union[Callable[[pydash.collections.T], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T]

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

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.

Returns *collection*

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: Dict[pydash.collections.T, pydash.collections.T2]*, *iteratee: Union[Callable[[pydash.collections.T2, pydash.collections.T, Dict[pydash.collections.T, pydash.collections.T2]], Any], int, str, List, Tuple, Dict]*) → Dict[pydash.collections.T, pydash.collections.T2]
pydash.collections.**for_each_right**(*collection: Dict[pydash.collections.T, pydash.collections.T2]*, *iteratee: Union[Callable[[pydash.collections.T2, pydash.collections.T], Any], int, str, List, Tuple, Dict]*) → Dict[pydash.collections.T, pydash.collections.T2]
pydash.collections.**for_each_right**(*collection: Dict[pydash.collections.T, pydash.collections.T2]*, *iteratee: Union[Callable[[pydash.collections.T2], Any], int, str, List, Tuple, Dict]*) → Dict[pydash.collections.T, pydash.collections.T2]
pydash.collections.**for_each_right**(*collection: List[pydash.collections.T]*, *iteratee: Union[Callable[[pydash.collections.T, int, List[pydash.collections.T]], Any], int, str, List, Tuple, Dict]*) → List[pydash.collections.T]
pydash.collections.**for_each_right**(*collection: List[pydash.collections.T]*, *iteratee: Union[Callable[[pydash.collections.T, int], Any], int, str, List, Tuple, Dict]*) → List[pydash.collections.T]
pydash.collections.**for_each_right**(*collection: List[pydash.collections.T]*, *iteratee: Union[Callable[[pydash.collections.T], Any], int, str, List, Tuple, Dict]*) → List[pydash.collections.T]

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

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.

Returns *collection*

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: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T], pydash.collections.T2]*) → Dict[pydash.collections.T2, List[pydash.collections.T]]
pydash.collections.**group_by**(*collection: Iterable[pydash.collections.T]*, *iteratee: Optional[Union[int, str, List, Tuple, Dict]] = None*) → Dict[Any, List[pydash.collections.T]]

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

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.

Returns Results of grouping by *iteratee*.

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: Union[Sequence, Dict]*, *target: Any*, *from_index: int = 0*) → bool

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** – Collection to iterate over.
- **target** – Target value to compare to.
- **from_index** – Offset to start search from.

Returns Whether *target* is in *collection*.

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: Iterable*, *path: Union[Hashable, List[Hashable]]*, **args: Any*, ***kwargs: Any*) → List[Any]

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** – Collection to iterate over.
- **path** – String path to method to invoke or callable to invoke for each element in *collection*.
- **args** – Arguments to pass to method call.
- **kwargs** – Keyword arguments to pass to method call.

Returns List of results of invoking method of each item.

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: Iterable[pydash.collections.T], iteratee: Callable[[pydash.collections.T], pydash.collections.T2]*) → Dict[pydash.collections.T2, pydash.collections.T]
pydash.collections.**key_by**(*collection: Iterable, iteratee: Optional[Union[int, str, List, Tuple, Dict]] = None*) → Dict

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

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.

Returns Results of indexing by *iteratee*.

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: Mapping[Any, pydash.collections.T2], iteratee: Callable[[pydash.collections.T2], pydash.collections.T3]*) → List[pydash.collections.T3]
pydash.collections.**map_**(*collection: Mapping[pydash.collections.T, pydash.collections.T2], iteratee: Callable[[pydash.collections.T2, pydash.collections.T], pydash.collections.T3]*) → List[pydash.collections.T3]
pydash.collections.**map_**(*collection: Mapping[pydash.collections.T, pydash.collections.T2], iteratee: Callable[[pydash.collections.T2, pydash.collections.T, Dict[pydash.collections.T, pydash.collections.T2]], pydash.collections.T3]*) → List[pydash.collections.T3]
pydash.collections.**map_**(*collection: Iterable[pydash.collections.T], iteratee: Callable[[pydash.collections.T], pydash.collections.T2]*) → List[pydash.collections.T2]
pydash.collections.**map_**(*collection: Iterable[pydash.collections.T], iteratee: Callable[[pydash.collections.T, int], pydash.collections.T2]*) → List[pydash.collections.T2]
pydash.collections.**map_**(*collection: Iterable[pydash.collections.T], iteratee: Callable[[pydash.collections.T, int, List[pydash.collections.T]], pydash.collections.T2]*) → List[pydash.collections.T2]
pydash.collections.**map_**(*collection: Iterable, iteratee: Optional[Union[int, str, List, Tuple, Dict]] = None*) → List

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 *matches()* style iteratee will return `True` for elements that have the properties of the given object, else `False`.

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.

Returns Mapped 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.**nest**(*collection: Iterable*, **properties: Any*) → Any

This method is like *group_by()* except that it supports nested grouping by multiple string *properties*. If only a single key is given, it is like calling `group_by(collection, prop)`.

Parameters

- **collection** – Collection to iterate over.
- ***properties** – Properties to nest by.

Returns Results of nested grouping by *properties*.

Example

```
>>> results = nest([{'shape': 'square', 'color': 'red', 'qty': 5},
↪          {'shape': 'square', 'color': 'blue', 'qty': 10},
↪    {'shape': 'square', 'color': 'orange', 'qty': 5},
↪'shape': 'circle', 'color': 'yellow', 'qty': 5},
↪'shape': 'circle', 'color': 'pink', 'qty': 10},
↪': 'oval', 'color': 'purple', 'qty': 5}],
↪')
>>> expected = {
↪': 5},
↪],
↪          'circle': {5: [{'shape': 'circle', 'color': 'yellow', 'qty': 5}],
↪ 'oval': {5: [{'shape': 'oval', 'color': 'purple', 'qty': 5}]}}
>>> results == expected
True
```

New in version 4.3.0.

pydash.collections.**order_by**(*collection: Mapping[Any, pydash.collections.T2]*, *keys: Iterable[Union[str, int]]*, *orders: Union[Iterable[bool], bool]*, *reverse: bool = False*) → List[pydash.collections.T2]
pydash.collections.**order_by**(*collection: Mapping[Any, pydash.collections.T2]*, *keys: Iterable[str]*, *orders: None = None*, *reverse: bool = False*) → List[pydash.collections.T2]
pydash.collections.**order_by**(*collection: Iterable[pydash.collections.T]*, *keys: Iterable[Union[str, int]]*, *orders: Union[Iterable[bool], bool]*, *reverse: bool = False*) → List[pydash.collections.T]
pydash.collections.**order_by**(*collection: Iterable[pydash.collections.T]*, *keys: Iterable[str]*, *orders: None = None*, *reverse: bool = False*) → List[pydash.collections.T]

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** – Collection to iterate over.
- **keys** – 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 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.

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

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: Mapping[pydash.collections.T, pydash.collections.T2]*, *predicate: Callable[[pydash.collections.T2, pydash.collections.T, Dict[pydash.collections.T, pydash.collections.T2]], Any]*) → List[List[pydash.collections.T2]]

pydash.collections.**partition**(*collection: Mapping[pydash.collections.T, pydash.collections.T2]*, *predicate: Callable[[pydash.collections.T2, pydash.collections.T], Any]*) → List[List[pydash.collections.T2]]
pydash.collections.**partition**(*collection: Mapping[Any, pydash.collections.T2]*, *predicate: Callable[[pydash.collections.T2], Any]*) → List[List[pydash.collections.T2]]
pydash.collections.**partition**(*collection: Mapping[Any, pydash.collections.T2]*, *predicate: Optional[Union[int, str, List, Tuple, Dict]] = None*) → List[List[pydash.collections.T2]]
pydash.collections.**partition**(*collection: Iterable[pydash.collections.T]*, *predicate: Callable[[pydash.collections.T, int, List[pydash.collections.T]], Any]*) → List[List[pydash.collections.T]]
pydash.collections.**partition**(*collection: Iterable[pydash.collections.T]*, *predicate: Callable[[pydash.collections.T, int], Any]*) → List[List[pydash.collections.T]]
pydash.collections.**partition**(*collection: Iterable[pydash.collections.T]*, *predicate: Callable[[pydash.collections.T], Any]*) → List[List[pydash.collections.T]]
pydash.collections.**partition**(*collection: Iterable[pydash.collections.T]*, *predicate: Optional[Union[int, str, List, Tuple, Dict]] = None*) → List[List[pydash.collections.T]]

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 *matches()* style predicate returns `True` for elements that have the properties of the given object, else `False`.

Parameters

- **collection** – Collection to iterate over.
- **predicate** – Predicate applied per iteration.

Returns List of grouped elements.

Example

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

New in version 1.1.0.

pydash.collections.**pluck**(*collection: Iterable*, *path: Union[Hashable, List[Hashable]]*) → List

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

Parameters

- **collection** – List of dicts.
- **path** – Collection's path to pluck

Returns Plucked 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: Mapping[pydash.collections.T, pydash.collections.T2], iteratee: Callable[[pydash.collections.T3, pydash.collections.T2, pydash.collections.T], pydash.collections.T3], accumulator: pydash.collections.T3*) → pydash.collections.T3

pydash.collections.**reduce_**(*collection: Mapping[Any, pydash.collections.T2], iteratee: Callable[[pydash.collections.T3, pydash.collections.T2], pydash.collections.T3], accumulator: pydash.collections.T3*) → pydash.collections.T3

pydash.collections.**reduce_**(*collection: Mapping, iteratee: Callable[[pydash.collections.T3], pydash.collections.T3], accumulator: pydash.collections.T3*) → pydash.collections.T3

pydash.collections.**reduce_**(*collection: Mapping[pydash.collections.T, pydash.collections.T2], iteratee: Callable[[pydash.collections.T2, pydash.collections.T2, pydash.collections.T], pydash.collections.T2], accumulator: None = None*) → pydash.collections.T2

pydash.collections.**reduce_**(*collection: Mapping[Any, pydash.collections.T2], iteratee: Callable[[pydash.collections.T2, pydash.collections.T2], pydash.collections.T2], accumulator: None = None*) → pydash.collections.T2

pydash.collections.**reduce_**(*collection: Mapping, iteratee: Callable[[pydash.collections.T], pydash.collections.T], accumulator: None = None*) → pydash.collections.T

pydash.collections.**reduce_**(*collection: Iterable[pydash.collections.T], iteratee: Callable[[pydash.collections.T2, pydash.collections.T, int], pydash.collections.T2], accumulator: pydash.collections.T2*) → pydash.collections.T2

pydash.collections.**reduce_**(*collection: Iterable[pydash.collections.T], iteratee: Callable[[pydash.collections.T2, pydash.collections.T], pydash.collections.T2], accumulator: pydash.collections.T2*) → pydash.collections.T2

pydash.collections.**reduce_**(*collection: Iterable, iteratee: Callable[[pydash.collections.T2], pydash.collections.T2], accumulator: pydash.collections.T2*) → pydash.collections.T2

pydash.collections.**reduce_**(*collection: Iterable[pydash.collections.T], iteratee: Callable[[pydash.collections.T, pydash.collections.T, int], pydash.collections.T], accumulator: None = None*) → pydash.collections.T

pydash.collections.**reduce_**(*collection: Iterable[pydash.collections.T], iteratee: Callable[[pydash.collections.T, pydash.collections.T], pydash.collections.T], accumulator: None = None*) → pydash.collections.T

pydash.collections.**reduce_**(*collection: Iterable, iteratee: Callable[[pydash.collections.T], pydash.collections.T], accumulator: None = None*) → pydash.collections.T

pydash.collections.**reduce_**(*collection: Iterable[pydash.collections.T]*, *iteratee: None = None*, *accumulator: Optional[pydash.collections.T] = None*) → pydash.collections.T

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** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.
- **accumulator** – Initial value of aggregator. Default is to use the result of the first iteration.

Returns Accumulator object containing results of reduction.

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: Mapping[pydash.collections.T, pydash.collections.T2]*, *iteratee: Callable[[pydash.collections.T3, pydash.collections.T2, pydash.collections.T], pydash.collections.T3]*, *accumulator: pydash.collections.T3*) → pydash.collections.T3

pydash.collections.**reduce_right**(*collection: Mapping[Any, pydash.collections.T2]*, *iteratee: Callable[[pydash.collections.T3, pydash.collections.T2], pydash.collections.T3]*, *accumulator: pydash.collections.T3*) → pydash.collections.T3

pydash.collections.**reduce_right**(*collection: Mapping*, *iteratee: Callable[[pydash.collections.T3], pydash.collections.T3]*, *accumulator: pydash.collections.T3*) → pydash.collections.T3

pydash.collections.**reduce_right**(*collection: Mapping[pydash.collections.T, pydash.collections.T2]*, *iteratee: Callable[[pydash.collections.T2, pydash.collections.T2, pydash.collections.T], pydash.collections.T2]*, *accumulator: None = None*) → pydash.collections.T2

pydash.collections.**reduce_right**(*collection: Mapping[Any, pydash.collections.T2]*, *iteratee: Callable[[pydash.collections.T2, pydash.collections.T2], pydash.collections.T2]*, *accumulator: None = None*) → pydash.collections.T2

pydash.collections.**reduce_right**(*collection: Mapping*, *iteratee: Callable[[pydash.collections.T], pydash.collections.T]*, *accumulator: None = None*) → pydash.collections.T

pydash.collections.**reduce_right**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T2, pydash.collections.T, int], pydash.collections.T2]*, *accumulator: pydash.collections.T2*) → pydash.collections.T2

pydash.collections.**reduce_right**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T2, pydash.collections.T], pydash.collections.T2]*, *accumulator: pydash.collections.T2*) → pydash.collections.T2

pydash.collections.**reduce_right**(*collection: Iterable*, *iteratee: Callable[[pydash.collections.T2], pydash.collections.T2]*, *accumulator: pydash.collections.T2*) → pydash.collections.T2

pydash.collections.**reduce_right**(*collection: Iterable[pydash.collections.T], iteratee: Callable[[pydash.collections.T, pydash.collections.T, int], pydash.collections.T], accumulator: None = None*) → pydash.collections.T
pydash.collections.**reduce_right**(*collection: Iterable[pydash.collections.T], iteratee: Callable[[pydash.collections.T, pydash.collections.T], pydash.collections.T], accumulator: None = None*) → pydash.collections.T
pydash.collections.**reduce_right**(*collection: Iterable, iteratee: Callable[[pydash.collections.T], pydash.collections.T], accumulator: None = None*) → pydash.collections.T
pydash.collections.**reduce_right**(*collection: Iterable[pydash.collections.T], iteratee: None = None, accumulator: Optional[pydash.collections.T] = None*) → pydash.collections.T

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

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.
- **accumulator** – Initial value of aggregator. Default is to use the result of the first iteration.

Returns Accumulator object containing results of reduction.

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: Mapping[pydash.collections.T, pydash.collections.T2], iteratee: Callable[[pydash.collections.T3, pydash.collections.T2, pydash.collections.T], pydash.collections.T3], accumulator: pydash.collections.T3, from_right: bool = False*) → List[pydash.collections.T3]
pydash.collections.**reductions**(*collection: Mapping[Any, pydash.collections.T2], iteratee: Callable[[pydash.collections.T3, pydash.collections.T2], pydash.collections.T3], accumulator: pydash.collections.T3, from_right: bool = False*) → List[pydash.collections.T3]
pydash.collections.**reductions**(*collection: Mapping, iteratee: Callable[[pydash.collections.T3], pydash.collections.T3], accumulator: pydash.collections.T3, from_right: bool = False*) → List[pydash.collections.T3]
pydash.collections.**reductions**(*collection: Mapping[pydash.collections.T, pydash.collections.T2], iteratee: Callable[[pydash.collections.T2, pydash.collections.T2, pydash.collections.T], pydash.collections.T2], accumulator: None = None, from_right: bool = False*) → List[pydash.collections.T2]
pydash.collections.**reductions**(*collection: Mapping[Any, pydash.collections.T2], iteratee: Callable[[pydash.collections.T2, pydash.collections.T2], pydash.collections.T2], accumulator: None = None, from_right: bool = False*) → List[pydash.collections.T2]

pydash.collections.**reductions**(*collection: Mapping*, *iteratee: Callable[[pydash.collections.T], pydash.collections.T]*, *accumulator: None = None*, *from_right: bool = False*) → List[pydash.collections.T]
pydash.collections.**reductions**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T2, pydash.collections.T, int], pydash.collections.T2]*, *accumulator: pydash.collections.T2*, *from_right: bool = False*) → List[pydash.collections.T2]
pydash.collections.**reductions**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T2, pydash.collections.T], pydash.collections.T2]*, *accumulator: pydash.collections.T2*, *from_right: bool = False*) → List[pydash.collections.T2]
pydash.collections.**reductions**(*collection: Iterable*, *iteratee: Callable[[pydash.collections.T2], pydash.collections.T2]*, *accumulator: pydash.collections.T2*, *from_right: bool = False*) → List[pydash.collections.T2]
pydash.collections.**reductions**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T, pydash.collections.T, int], pydash.collections.T]*, *accumulator: None = None*, *from_right: bool = False*) → List[pydash.collections.T]
pydash.collections.**reductions**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T, pydash.collections.T], pydash.collections.T]*, *accumulator: None = None*, *from_right: bool = False*) → List[pydash.collections.T]
pydash.collections.**reductions**(*collection: Iterable*, *iteratee: Callable[[pydash.collections.T], pydash.collections.T]*, *accumulator: None = None*, *from_right: bool = False*) → List[pydash.collections.T]
pydash.collections.**reductions**(*collection: Iterable[pydash.collections.T]*, *iteratee: None = None*, *accumulator: Optional[pydash.collections.T] = None*, *from_right: bool = False*) → List[pydash.collections.T]

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

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.
- **accumulator** – Initial value of aggregator. Default is to use the result of the first iteration.

Returns Results of each reduction operation.

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: Mapping[pydash.collections.T, pydash.collections.T2]*, *iteratee: Callable[[pydash.collections.T3, pydash.collections.T2, pydash.collections.T], pydash.collections.T3]*, *accumulator: pydash.collections.T3*) → List[pydash.collections.T3]

pydash.collections.**reductions_right**(*collection: Mapping[Any, pydash.collections.T2]*, *iteratee: Callable[[pydash.collections.T3, pydash.collections.T2], pydash.collections.T3]*, *accumulator: pydash.collections.T3*) → List[pydash.collections.T3]
pydash.collections.**reductions_right**(*collection: Mapping*, *iteratee: Callable[[pydash.collections.T3], pydash.collections.T3]*, *accumulator: pydash.collections.T3*) → List[pydash.collections.T3]
pydash.collections.**reductions_right**(*collection: Mapping[pydash.collections.T, pydash.collections.T2]*, *iteratee: Callable[[pydash.collections.T2, pydash.collections.T2, pydash.collections.T], pydash.collections.T2]*, *accumulator: None = None*) → List[pydash.collections.T2]
pydash.collections.**reductions_right**(*collection: Mapping[Any, pydash.collections.T2]*, *iteratee: Callable[[pydash.collections.T2, pydash.collections.T2], pydash.collections.T2]*, *accumulator: None = None*) → List[pydash.collections.T2]
pydash.collections.**reductions_right**(*collection: Mapping*, *iteratee: Callable[[pydash.collections.T], pydash.collections.T]*, *accumulator: None = None*) → List[pydash.collections.T]
pydash.collections.**reductions_right**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T2, pydash.collections.T, int], pydash.collections.T2]*, *accumulator: pydash.collections.T2*) → List[pydash.collections.T2]
pydash.collections.**reductions_right**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T2, pydash.collections.T], pydash.collections.T2]*, *accumulator: pydash.collections.T2*) → List[pydash.collections.T2]
pydash.collections.**reductions_right**(*collection: Iterable*, *iteratee: Callable[[pydash.collections.T2], pydash.collections.T2]*, *accumulator: pydash.collections.T2*) → List[pydash.collections.T2]
pydash.collections.**reductions_right**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T, pydash.collections.T, int], pydash.collections.T]*, *accumulator: None = None*) → List[pydash.collections.T]
pydash.collections.**reductions_right**(*collection: Iterable[pydash.collections.T]*, *iteratee: Callable[[pydash.collections.T, pydash.collections.T], pydash.collections.T]*, *accumulator: None = None*) → List[pydash.collections.T]
pydash.collections.**reductions_right**(*collection: Iterable*, *iteratee: Callable[[pydash.collections.T], pydash.collections.T]*, *accumulator: None = None*) → List[pydash.collections.T]
pydash.collections.**reductions_right**(*collection: Iterable[pydash.collections.T]*, *iteratee: None = None*, *accumulator: Optional[pydash.collections.T] = None*) → List[pydash.collections.T]

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

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.
- **accumulator** – Initial value of aggregator. Default is to use the result of the first iteration.

Returns Results of each reduction operation.

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: Mapping[pydash.collections.T, pydash.collections.T2], predicate: Optional[Union[Callable[[pydash.collections.T2, pydash.collections.T, Dict[pydash.collections.T, pydash.collections.T2]], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T2]
pydash.collections.**reject**(*collection: Mapping[pydash.collections.T, pydash.collections.T2], predicate: Optional[Union[Callable[[pydash.collections.T2, pydash.collections.T], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T2]
pydash.collections.**reject**(*collection: Mapping[Any, pydash.collections.T2], predicate: Optional[Union[Callable[[pydash.collections.T2], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T2]
pydash.collections.**reject**(*collection: Iterable[pydash.collections.T], predicate: Optional[Union[Callable[[pydash.collections.T, int, List[pydash.collections.T]], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T]
pydash.collections.**reject**(*collection: Iterable[pydash.collections.T], predicate: Optional[Union[Callable[[pydash.collections.T, int], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T]
pydash.collections.**reject**(*collection: Iterable[pydash.collections.T], predicate: Optional[Union[Callable[[pydash.collections.T], Any], int, str, List, Tuple, Dict]] = None*) → List[pydash.collections.T]

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

Parameters

- **collection** – Collection to iterate over.
- **predicate** – Predicate applied per iteration.

Returns Rejected elements of *collection*.

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: Sequence[pydash.collections.T]*) → pydash.collections.T

Retrieves a random element from a given *collection*.

Parameters **collection** – Collection to iterate over.

Returns Random element from the given collection.

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: Sequence[pydash.collections.T]*, *n: Optional[int] = None*) → List[pydash.collections.T]

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

Parameters

- **collection** – Collection to iterate over.
- **n** – Number of random samples to return.

Returns List of *n* sampled collection values.

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: Mapping[Any, pydash.collections.T]*) → List[pydash.collections.T]
pydash.collections.**shuffle**(*collection: Iterable[pydash.collections.T]*) → List[pydash.collections.T]

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

Parameters **collection** – Collection to iterate over.

Returns Shuffled list of values.

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: Sized*) → int

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

Parameters **collection** – Collection to iterate over.

Returns Collection length.

Example

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

New in version 1.0.0.

pydash.collections.**some**(*collection: Iterable[pydash.collections.T]*, *predicate: Optional[Callable[[pydash.collections.T], Any]] = None*) → bool

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 *matches()* style predicate will return `True` for elements that have the properties of the given object, else `False`.

Parameters

- **collection** – Collection to iterate over.
- **predicate** – Predicate applied per iteration.

Returns Whether any of the elements are truthy.

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: Mapping[Any, pydash.collections.T2]*, *iteratee: Optional[Union[Callable[[pydash.collections.T2], Any], int, str, List, Tuple, Dict]] = None*, *reverse: bool = False*) → List[pydash.collections.T2]

pydash.collections.**sort_by**(*collection: Iterable[pydash.collections.T]*, *iteratee: Optional[Union[Callable[[pydash.collections.T], Any], int, str, List, Tuple, Dict]] = None*, *reverse: bool = False*) → List[pydash.collections.T]

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

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.
- **reverse** – Whether to reverse the sort. Defaults to `False`.

Returns Sorted 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.

5.1.5 Functions

Functions that wrap other functions.

New in version 1.0.0.

pydash.functions.**after**(*func: Callable[[pydash.functions.P], pydash.functions.T]*, *n: SupportsInt*) → pydash.functions.After[pydash.functions.P, pydash.functions.T]

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

Parameters

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

Returns Function wrapped in an `After` context.

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: Callable[[...], pydash.functions.T]*, *n: Optional[SupportsInt]*) → pydash.functions.Ary[pydash.functions.T]

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 to cap arguments for.
- **n** – Number of arguments to accept.

Returns Function wrapped in an `Ary` context.

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: Callable[[pydash.functions.P], pydash.functions.T]*, *n: SupportsInt*) → pydash.functions.Before[pydash.functions.P, pydash.functions.T]

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

Parameters

- **func** – Function to execute.
- **n** – Number of times *func* may be executed.

Returns Function wrapped in an `Before` context.

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)
>>> before_func(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: Callable[[pydash.functions.T], Any]*) → Callable[[Iterable[pydash.functions.T]], bool]

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(s) to conjoin.

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

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: Callable[[pydash.functions.T1], pydash.functions.T]*, *arity: Optional[int] = None*) → pydash.functions.CurryOne[pydash.functions.T1, pydash.functions.T]
pydash.functions.**curry**(*func: Callable[[pydash.functions.T1, pydash.functions.T2], pydash.functions.T]*, *arity: Optional[int] = None*) → pydash.functions.CurryTwo[pydash.functions.T1, pydash.functions.T2, pydash.functions.T]
pydash.functions.**curry**(*func: Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3], pydash.functions.T]*, *arity: Optional[int] = None*) → pydash.functions.CurryThree[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3, pydash.functions.T]
pydash.functions.**curry**(*func: Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3, pydash.functions.T4], pydash.functions.T]*, *arity: Optional[int] = None*) → pydash.functions.CurryFour[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3, pydash.functions.T4, pydash.functions.T]
pydash.functions.**curry**(*func: Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3, pydash.functions.T4, pydash.functions.T5], pydash.functions.T]*, *arity: Optional[int] = None*) → pydash.functions.CurryFive[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3, pydash.functions.T4, pydash.functions.T5, pydash.functions.T]

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 to curry.
- **arity** – 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.

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: Callable[[pydash.functions.T1], pydash.functions.T]*, *arity: Optional[int] = None*) → pydash.functions.CurryRightOne[pydash.functions.T1, pydash.functions.T]
pydash.functions.**curry_right**(*func: Callable[[pydash.functions.T1, pydash.functions.T2], pydash.functions.T]*, *arity: Optional[int] = None*) → pydash.functions.CurryRightTwo[pydash.functions.T2, pydash.functions.T1, pydash.functions.T]
pydash.functions.**curry_right**(*func: Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3], pydash.functions.T]*, *arity: Optional[int] = None*) → pydash.functions.CurryRightThree[pydash.functions.T3, pydash.functions.T2, pydash.functions.T1, pydash.functions.T]
pydash.functions.**curry_right**(*func: Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3, pydash.functions.T4], pydash.functions.T]*, *arity: Optional[int] = None*) → pydash.functions.CurryRightFour[pydash.functions.T4, pydash.functions.T3, pydash.functions.T2, pydash.functions.T1, pydash.functions.T]
pydash.functions.**curry_right**(*func: Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3, pydash.functions.T4, pydash.functions.T5], pydash.functions.T]*) → pydash.functions.CurryRightFive[pydash.functions.T5, pydash.functions.T4, pydash.functions.T3, pydash.functions.T2, pydash.functions.T1, pydash.functions.T]

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

Parameters

- **func** – Function to curry.
- **arity** – 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.

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: Callable[[pydash.functions.P], pydash.functions.T]*, *wait: int*, *max_wait: Union[int, typing_extensions.Literal[False]] = False*) → pydash.functions.Debounce[pydash.functions.P, pydash.functions.T]

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

Returns Function wrapped in a `Debounce` context.

New in version 1.0.0.

pydash.functions.**delay**(*func: Callable[[pydash.functions.P], pydash.functions.T]*, *wait: int*, **args: P.args*, ***kwargs: P.kwargs*) → pydash.functions.T

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

Parameters

- **func** – Function to execute.
- **wait** – Milliseconds to wait before executing *func*.
- ***args** – Arguments to pass to *func*.
- ****kwargs** – Keyword arguments to pass to *func*.

Returns Return from *func*.

New in version 1.0.0.

pydash.functions.**disjoin**(**funcs: Callable[[pydash.functions.T], Any]*) → pydash.functions.Disjoin[pydash.functions.T]

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(s) to disjoin.

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

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: Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3, pydash.functions.T4, pydash.functions.T5], pydash.functions.T]*) → Callable[[pydash.functions.T5, pydash.functions.T4, pydash.functions.T3, pydash.functions.T2, pydash.functions.T1], pydash.functions.T]

pydash.functions.**flip**(*func: Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3, pydash.functions.T4], pydash.functions.T]*) → Callable[[pydash.functions.T4, pydash.functions.T3, pydash.functions.T2, pydash.functions.T1], pydash.functions.T]

pydash.functions.**flip**(*func: Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3], pydash.functions.T]*) → Callable[[pydash.functions.T3, pydash.functions.T2, pydash.functions.T1], pydash.functions.T]
pydash.functions.**flip**(*func: Callable[[pydash.functions.T1, pydash.functions.T2], pydash.functions.T]*) → Callable[[pydash.functions.T2, pydash.functions.T1], pydash.functions.T]
pydash.functions.**flip**(*func: Callable[[pydash.functions.T1], pydash.functions.T]*) → Callable[[pydash.functions.T1], pydash.functions.T]

Creates a function that invokes the method with arguments reversed.

Parameters **func** – Function to flip arguments for.

Returns Function wrapped in a `Flip` context.

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**(*func1: Callable[[pydash.functions.P], pydash.functions.T2], func2: Callable[[pydash.functions.T2], pydash.functions.T3], func3: Callable[[pydash.functions.T3], pydash.functions.T4], func4: Callable[[pydash.functions.T4], pydash.functions.T5], func5: Callable[[pydash.functions.T5], pydash.functions.T]*) → pydash.functions.Flow[pydash.functions.P, pydash.functions.T]
pydash.functions.**flow**(*func1: Callable[[pydash.functions.P], pydash.functions.T2], func2: Callable[[pydash.functions.T2], pydash.functions.T3], func3: Callable[[pydash.functions.T3], pydash.functions.T4], func4: Callable[[pydash.functions.T4], pydash.functions.T]*) → pydash.functions.Flow[pydash.functions.P, pydash.functions.T]
pydash.functions.**flow**(*func1: Callable[[pydash.functions.P], pydash.functions.T2], func2: Callable[[pydash.functions.T2], pydash.functions.T3], func3: Callable[[pydash.functions.T3], pydash.functions.T]*) → pydash.functions.Flow[pydash.functions.P, pydash.functions.T]
pydash.functions.**flow**(*func1: Callable[[pydash.functions.P], pydash.functions.T2], func2: Callable[[pydash.functions.T2], pydash.functions.T]*) → pydash.functions.Flow[pydash.functions.P, pydash.functions.T]
pydash.functions.**flow**(*func1: Callable[[pydash.functions.P], pydash.functions.T]*) → pydash.functions.Flow[pydash.functions.P, pydash.functions.T]

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(s) to compose.

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

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**(*func5: Callable[[pydash.functions.T4], pydash.functions.T]*, *func4: Callable[[pydash.functions.T3], pydash.functions.T4]*, *func3: Callable[[pydash.functions.T2], pydash.functions.T3]*, *func2: Callable[[pydash.functions.T1], pydash.functions.T2]*, *func1: Callable[[pydash.functions.P], pydash.functions.T1]*) → pydash.functions.Flow[pydash.functions.P, pydash.functions.T]
pydash.functions.**flow_right**(*func4: Callable[[pydash.functions.T3], pydash.functions.T]*, *func3: Callable[[pydash.functions.T2], pydash.functions.T3]*, *func2: Callable[[pydash.functions.T1], pydash.functions.T2]*, *func1: Callable[[pydash.functions.P], pydash.functions.T1]*) → pydash.functions.Flow[pydash.functions.P, pydash.functions.T]
pydash.functions.**flow_right**(*func3: Callable[[pydash.functions.T2], pydash.functions.T]*, *func2: Callable[[pydash.functions.T1], pydash.functions.T2]*, *func1: Callable[[pydash.functions.P], pydash.functions.T1]*) → pydash.functions.Flow[pydash.functions.P, pydash.functions.T]
pydash.functions.**flow_right**(*func2: Callable[[pydash.functions.T1], pydash.functions.T]*, *func1: Callable[[pydash.functions.P], pydash.functions.T1]*) → pydash.functions.Flow[pydash.functions.P, pydash.functions.T]
pydash.functions.**flow_right**(*func1: Callable[[pydash.functions.P], pydash.functions.T]*) → pydash.functions.Flow[pydash.functions.P, pydash.functions.T]

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(s) to compose.

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

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: Callable[[pydash.functions.T], pydash.functions.T]*) → pydash.functions.Iterated[pydash.functions.T]

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 to iterate.

Returns Function wrapped in a `Iterated` context.

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: Callable[[pydash.functions.P], pydash.functions.T]*) → pydash.functions.Juxtapose[pydash.functions.P, pydash.functions.T]

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

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

Returns Function wrapped in a `Juxtapose` context.

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: Callable[[pydash.functions.P], Any]*) → pydash.functions.Negate[pydash.functions.P]

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 to negate execute.

Returns Function wrapped in a `Negate` context.

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: Callable[[pydash.functions.P], pydash.functions.T]*) → pydash.functions.Once[pydash.functions.P, pydash.functions.T]

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 to execute.

Returns Function wrapped in a `Once` context.

Example

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

New in version 1.0.0.

pydash.functions.**over_args**(*func: Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3, pydash.functions.T4, pydash.functions.T5], pydash.functions.T]*, *transform_one: Callable[[pydash.functions.T1], pydash.functions.T1]*, *transform_two: Callable[[pydash.functions.T2], pydash.functions.T2]*, *transform_three: Callable[[pydash.functions.T3], pydash.functions.T3]*, *transform_four: Callable[[pydash.functions.T4], pydash.functions.T4]*, *transform_five: Callable[[pydash.functions.T5], pydash.functions.T5]*) → Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3, pydash.functions.T4, pydash.functions.T5], pydash.functions.T]

pydash.functions.**over_args**(*func: Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3, pydash.functions.T4], pydash.functions.T]*, *transform_one: Callable[[pydash.functions.T1], pydash.functions.T1]*, *transform_two: Callable[[pydash.functions.T2], pydash.functions.T2]*, *transform_three: Callable[[pydash.functions.T3], pydash.functions.T3]*, *transform_four: Callable[[pydash.functions.T4], pydash.functions.T4]*) → Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3, pydash.functions.T4], pydash.functions.T]

pydash.functions.**over_args**(*func: Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3], pydash.functions.T]*, *transform_one: Callable[[pydash.functions.T1], pydash.functions.T1]*, *transform_two: Callable[[pydash.functions.T2], pydash.functions.T2]*, *transform_three: Callable[[pydash.functions.T3], pydash.functions.T3]*) → Callable[[pydash.functions.T1, pydash.functions.T2, pydash.functions.T3], pydash.functions.T]

pydash.functions.**over_args**(*func: Callable[[pydash.functions.T1, pydash.functions.T2], pydash.functions.T], transform_one: Callable[[pydash.functions.T1], pydash.functions.T1], transform_two: Callable[[pydash.functions.T2], pydash.functions.T2]*) → Callable[[pydash.functions.T1, pydash.functions.T2], pydash.functions.T]
pydash.functions.**over_args**(*func: Callable[[pydash.functions.T1], pydash.functions.T], transform_one: Callable[[pydash.functions.T1], pydash.functions.T1]*) → Callable[[pydash.functions.T1], pydash.functions.T]

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

Parameters

- **func** – Function to wrap.
- ***transforms** – Functions to transform arguments, specified as individual functions or lists of functions.

Returns Function wrapped in a `OverArgs` context.

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: Callable[[...], pydash.functions.T], *args: Any, **kwargs: Any*) → pydash.functions.Partial[pydash.functions.T]

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

Parameters

- **func** – Function to execute.
- ***args** – Partial arguments to prepend to function call.
- ****kwargs** – Partial keyword arguments to bind to function call.

Returns Function wrapped in a `Partial` context.

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: Callable[[...], pydash.functions.T]*, **args: Any*, ***kwargs: Any*) → pydash.functions.Partial[pydash.functions.T]

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

Parameters

- **func** – Function to execute.
- ***args** – Partial arguments to append to function call.
- ****kwargs** – Partial keyword arguments to bind to function call.

Returns Function wrapped in a `Partial` context.

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: Callable[[pydash.functions.P], pydash.functions.T]*, **indexes: int*) → pydash.functions.Rearg[pydash.functions.P, pydash.functions.T]

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 to rearrange arguments for.
- ***indexes** – The arranged argument indexes.

Returns Function wrapped in a `Rearg` context.

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: Callable[[...], pydash.functions.T]*) → pydash.functions.Spread[pydash.functions.T]

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

Parameters **func** – Function to spread.

Returns Function wrapped in a `Spread` context.

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: Callable[[pydash.functions.P], pydash.functions.T]*, *wait: int*) → pydash.functions.Throttle[pydash.functions.P, pydash.functions.T]

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 to throttle.
- **wait** – Milliseconds to wait before calling *func* again.

Returns Results of last *func* call.

New in version 1.0.0.

pydash.functions.**unary**(*func: Callable[[...], pydash.functions.T]*) → pydash.functions.Ary[pydash.functions.T]

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

Parameters **func** – Function to cap arguments for.

Returns Function wrapped in an `Ary` context.

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: pydash.functions.T1*, *func: Callable[[pydash.functions.T1, pydash.functions.P], pydash.functions.T]*) → pydash.functions.Partial[pydash.functions.T]

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** – Value provided as first argument to function call.
- **func** – Function to execute.

Returns Function wrapped in a `Partial` context.

Example

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

New in version 1.0.0.

5.1.6 Numerical

Numerical/mathematical related functions.

New in version 2.1.0.

pydash.numerical.**add**(*a: SupportsAdd[T, T2]*, *b: pydash.numerical.T*) → pydash.numerical.T2
pydash.numerical.**add**(*a: pydash.numerical.T*, *b: SupportsAdd[T, T2]*) → pydash.numerical.T2
Adds two numbers.

Parameters

- **a** – First number to add.
- **b** – 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: Union[float, int, Decimal]*, *precision: int = 0*) → float
Round number up to precision.

Parameters

- **x** – Number to round up.
- **precision** – Rounding precision. Defaults to `0`.

Returns Number rounded up.

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: pydash.numerical.NumT*, *lower: pydash.numerical.NumT2*, *upper: Optional[pydash.numerical.NumT3] = None*) → Union[pydash.numerical.NumT, pydash.numerical.NumT2, pydash.numerical.NumT3]

Clamps number within the inclusive lower and upper bounds.

Parameters

- **x** – Number to clamp.
- **lower** – Lower bound.
- **upper** – 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: Optional[Union[float, int, Decimal]]*, *divisor: Optional[Union[float, int, Decimal]]*) → float

Divide two numbers.

Parameters

- **dividend** – The first number in a division.
- **divisor** – The second number in a division.

Returns Returns the quotient.

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: Union[float, int, Decimal]*, *precision: int = 0*) → float

Round number down to precision.

Parameters

- **x** – Number to round down.
- **precision** – Rounding precision. Defaults to `0`.

Returns Number rounded down.

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: Mapping[Any, SupportsRichComparisonT]*, *default: pydash.helpers.Unset = UNSET*) → SupportsRichComparisonT

pydash.numerical.**max_**(*collection: Mapping[Any, SupportsRichComparisonT]*, *default: pydash.numerical.T*) → Union[SupportsRichComparisonT, pydash.numerical.T]

pydash.numerical.**max_**(*collection: Iterable[SupportsRichComparisonT]*, *default: pydash.helpers.Unset = UNSET*) → SupportsRichComparisonT

pydash.numerical.**max_**(*collection: Iterable[SupportsRichComparisonT]*, *default: pydash.numerical.T*) → Union[SupportsRichComparisonT, pydash.numerical.T]

Retrieves the maximum value of a *collection*.

Parameters

- **collection** – Collection to iterate over.
- **default** – Value to return if *collection* is empty.

Returns Maximum value.

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 iteratee support to *max_by()*.

pydash.numerical.**max_by**(*collection: Mapping[Any, SupportsRichComparisonT]*, *iteratee: None = None*, *default: pydash.helpers.Unset = UNSET*) → SupportsRichComparisonT
pydash.numerical.**max_by**(*collection: Mapping[Any, pydash.numerical.T2]*, *iteratee: Callable[[pydash.numerical.T2], SupportsRichComparisonT]*, *default: pydash.helpers.Unset = UNSET*) → pydash.numerical.T2
pydash.numerical.**max_by**(*collection: Mapping[Any, pydash.numerical.T2]*, *iteratee: Callable[[pydash.numerical.T2], SupportsRichComparisonT]*, ***, *default: pydash.numerical.T*) → Union[pydash.numerical.T2, pydash.numerical.T]
pydash.numerical.**max_by**(*collection: Mapping[Any, SupportsRichComparisonT]*, *iteratee: None = None*, ***, *default: pydash.numerical.T*) → Union[SupportsRichComparisonT, pydash.numerical.T]
pydash.numerical.**max_by**(*collection: Iterable[SupportsRichComparisonT]*, *iteratee: None = None*, *default: pydash.helpers.Unset = UNSET*) → SupportsRichComparisonT
pydash.numerical.**max_by**(*collection: Iterable[pydash.numerical.T2]*, *iteratee: Callable[[pydash.numerical.T2], SupportsRichComparisonT]*, *default: pydash.helpers.Unset = UNSET*) → pydash.numerical.T2
pydash.numerical.**max_by**(*collection: Iterable[pydash.numerical.T2]*, *iteratee: Callable[[pydash.numerical.T2], SupportsRichComparisonT]*, ***, *default: pydash.numerical.T*) → Union[pydash.numerical.T2, pydash.numerical.T]
pydash.numerical.**max_by**(*collection: Iterable[SupportsRichComparisonT]*, *iteratee: None = None*, ***, *default: pydash.numerical.T*) → Union[SupportsRichComparisonT, pydash.numerical.T]
pydash.numerical.**max_by**(*collection: Iterable[pydash.numerical.T]*, *iteratee: Union[int, str, List, Tuple, Dict]*, *default: pydash.helpers.Unset = UNSET*) → pydash.numerical.T
pydash.numerical.**max_by**(*collection: Iterable[pydash.numerical.T]*, *iteratee: Union[int, str, List, Tuple, Dict]*, *default: pydash.numerical.T2*) → Union[pydash.numerical.T, pydash.numerical.T2]

Retrieves the maximum value of a *collection*.

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.
- **default** – Value to return if *collection* is empty.

Returns Maximum value.

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: Mapping[Any, SupportsAdd[int, t.Any]]*) → float
pydash.numerical.**mean**(*collection: Iterable[SupportsAdd[int, t.Any]]*) → float

Calculate arithmetic mean of each element in *collection*.

Parameters **collection** – Collection to process.

Returns Result of mean.

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: Mapping[pydash.numerical.T, pydash.numerical.T2], iteratee: Callable[[pydash.numerical.T2, pydash.numerical.T, Dict[pydash.numerical.T, pydash.numerical.T2]], SupportsAdd[int, t.Any]]*) → float
pydash.numerical.**mean_by**(*collection: Mapping[pydash.numerical.T, pydash.numerical.T2], iteratee: Callable[[pydash.numerical.T2, pydash.numerical.T], SupportsAdd[int, t.Any]]*) → float
pydash.numerical.**mean_by**(*collection: Mapping[Any, pydash.numerical.T2], iteratee: Callable[[pydash.numerical.T2], SupportsAdd[int, t.Any]]*) → float
pydash.numerical.**mean_by**(*collection: Iterable[pydash.numerical.T], iteratee: Callable[[pydash.numerical.T, int, List[pydash.numerical.T]], SupportsAdd[int, t.Any]]*) → float
pydash.numerical.**mean_by**(*collection: Iterable[pydash.numerical.T], iteratee: Callable[[pydash.numerical.T, int], SupportsAdd[int, t.Any]]*) → float
pydash.numerical.**mean_by**(*collection: Iterable[pydash.numerical.T], iteratee: Callable[[pydash.numerical.T], SupportsAdd[int, t.Any]]*) → float
pydash.numerical.**mean_by**(*collection: Mapping[Any, SupportsAdd[int, t.Any]], iteratee: None = None*) → float
pydash.numerical.**mean_by**(*collection: Iterable[SupportsAdd[int, t.Any]], iteratee: None = None*) → float

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

Parameters

- **collection** – Collection to process.
- **iteratee** – Iteratee applied per iteration.

Returns Result of mean.

Example

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

New in version 4.0.0.

pydash.numerical.**median**(*collection: Mapping[pydash.numerical.T, pydash.numerical.T2]*, *iteratee: Callable[[pydash.numerical.T2, pydash.numerical.T, Dict[pydash.numerical.T, pydash.numerical.T2]], Union[float, int, Decimal]]*) → Union[float, int]
pydash.numerical.**median**(*collection: Mapping[pydash.numerical.T, pydash.numerical.T2]*, *iteratee: Callable[[pydash.numerical.T2, pydash.numerical.T], Union[float, int, Decimal]]*) → Union[float, int]
pydash.numerical.**median**(*collection: Mapping[Any, pydash.numerical.T2]*, *iteratee: Callable[[pydash.numerical.T2], Union[float, int, Decimal]]*) → Union[float, int]
pydash.numerical.**median**(*collection: Iterable[pydash.numerical.T]*, *iteratee: Callable[[pydash.numerical.T, int, List[pydash.numerical.T]], Union[float, int, Decimal]]*) → Union[float, int]
pydash.numerical.**median**(*collection: Iterable[pydash.numerical.T]*, *iteratee: Callable[[pydash.numerical.T, int], Union[float, int, Decimal]]*) → Union[float, int]
pydash.numerical.**median**(*collection: Iterable[pydash.numerical.T]*, *iteratee: Callable[[pydash.numerical.T], Union[float, int, Decimal]]*) → Union[float, int]
pydash.numerical.**median**(*collection: Iterable[Union[float, int, Decimal]]*, *iteratee: None = None*) → Union[float, int]

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

Parameters

- **collection** – Collection to process.
- **iteratee** – Iteratee applied per iteration.

Returns Result of median.

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: Mapping[Any, SupportsRichComparisonT]*, *default: pydash.helpers.Unset = UNSET*) → SupportsRichComparisonT
pydash.numerical.**min_**(*collection: Mapping[Any, SupportsRichComparisonT]*, *default: pydash.numerical.T*) → Union[SupportsRichComparisonT, pydash.numerical.T]
pydash.numerical.**min_**(*collection: Iterable[SupportsRichComparisonT]*, *default: pydash.helpers.Unset = UNSET*) → SupportsRichComparisonT
pydash.numerical.**min_**(*collection: Iterable[SupportsRichComparisonT]*, *default: pydash.numerical.T*) → Union[SupportsRichComparisonT, pydash.numerical.T]

Retrieves the minimum value of a *collection*.

Parameters

- **collection** – Collection to iterate over.

- **default** – Value to return if *collection* is empty.

Returns Minimum value.

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 iteratee support to *min_by()*.

pydash.numerical.**min_by**(*collection: Mapping[Any, SupportsRichComparisonT]*, *iteratee: None = None*, *default: pydash.helpers.Unset = UNSET*) → SupportsRichComparisonT
pydash.numerical.**min_by**(*collection: Mapping[Any, pydash.numerical.T2]*, *iteratee: Callable[[pydash.numerical.T2], SupportsRichComparisonT]*, *default: pydash.helpers.Unset = UNSET*) → pydash.numerical.T2
pydash.numerical.**min_by**(*collection: Mapping[Any, pydash.numerical.T2]*, *iteratee: Callable[[pydash.numerical.T2], SupportsRichComparisonT]*, ***, *default: pydash.numerical.T*) → Union[pydash.numerical.T2, pydash.numerical.T]
pydash.numerical.**min_by**(*collection: Mapping[Any, SupportsRichComparisonT]*, *iteratee: None = None*, ***, *default: pydash.numerical.T*) → Union[SupportsRichComparisonT, pydash.numerical.T]
pydash.numerical.**min_by**(*collection: Iterable[SupportsRichComparisonT]*, *iteratee: None = None*, *default: pydash.helpers.Unset = UNSET*) → SupportsRichComparisonT
pydash.numerical.**min_by**(*collection: Iterable[pydash.numerical.T2]*, *iteratee: Callable[[pydash.numerical.T2], SupportsRichComparisonT]*, *default: pydash.helpers.Unset = UNSET*) → pydash.numerical.T2
pydash.numerical.**min_by**(*collection: Iterable[pydash.numerical.T2]*, *iteratee: Callable[[pydash.numerical.T2], SupportsRichComparisonT]*, ***, *default: pydash.numerical.T*) → Union[pydash.numerical.T2, pydash.numerical.T]
pydash.numerical.**min_by**(*collection: Iterable[SupportsRichComparisonT]*, *iteratee: None = None*, ***, *default: pydash.numerical.T*) → Union[SupportsRichComparisonT, pydash.numerical.T]
pydash.numerical.**min_by**(*collection: Iterable[pydash.numerical.T]*, *iteratee: Union[int, str, List, Tuple, Dict]*, *default: pydash.helpers.Unset = UNSET*) → pydash.numerical.T
pydash.numerical.**min_by**(*collection: Iterable[pydash.numerical.T]*, *iteratee: Union[int, str, List, Tuple, Dict]*, *default: pydash.numerical.T2*) → Union[pydash.numerical.T, pydash.numerical.T2]

Retrieves the minimum value of a *collection*.

Parameters

- **collection** – Collection to iterate over.
- **iteratee** – Iteratee applied per iteration.
- **default** – Value to return if *collection* is empty.

Returns Minimum value.

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: Sequence[SupportsAdd[int, t.Any]]*, *size: SupportsInt*) → List[float]
Calculate moving mean of each element of *array*.

Parameters

- **array** – List to process.
- **size** – Window size.

Returns Result of moving average.

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: pydash.types.SupportsMul[int, pydash.numerical.T2]*, *multiplicand: None*) → pydash.numerical.T2
pydash.numerical.**multiply**(*multiplier: None*, *multiplicand: pydash.types.SupportsMul[int, pydash.numerical.T2]*) → pydash.numerical.T2
pydash.numerical.**multiply**(*multiplier: None*, *multiplicand: None*) → int
pydash.numerical.**multiply**(*multiplier: pydash.types.SupportsMul[pydash.numerical.T, pydash.numerical.T2]*, *multiplicand: pydash.numerical.T*) → pydash.numerical.T2
pydash.numerical.**multiply**(*multiplier: pydash.numerical.T*, *multiplicand: pydash.types.SupportsMul[pydash.numerical.T, pydash.numerical.T2]*) → pydash.numerical.T2
Multiply two numbers.

Parameters

- **multiplier** – The first number in a multiplication.
- **multiplicand** – The second number in a multiplication.

Returns Returns the product.

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: int*, *n: int*) → Union[int, float]
pydash.numerical.**power**(*x: float*, *n: Union[int, float]*) → float
pydash.numerical.**power**(*x: List[int]*, *n: int*) → List[Union[int, float]]
pydash.numerical.**power**(*x: List[float]*, *n: List[Union[int, float]]*) → List[float]

Calculate exponentiation of *x* raised to the *n* power.

Parameters

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

Returns Result of calculation.

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: List[pydash.types.SupportsRound[Union[float, int, Decimal]]]*, *precision: int = 0*) → List[float]
pydash.numerical.**round_**(*x: pydash.types.SupportsRound[Union[float, int, Decimal]]*, *precision: int = 0*) → float

Round number to precision.

Parameters

- **x** – Number to round.
- **precision** – Rounding precision. Defaults to `0`.

Returns Rounded number.

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: Iterable[Decimal]*, *maximum: Decimal*) → List[Decimal]
pydash.numerical.**scale**(*array: Iterable[Union[float, int]]*, *maximum: Union[float, int]*) → List[float]
pydash.numerical.**scale**(*array: Iterable[Union[float, int, Decimal]]*, *maximum: int = 1*) → List[float]

Scale list of value to a maximum number.

Parameters

- **array** – Numbers to scale.
- **maximum** – Maximum scale value.

Returns Scaled numbers.

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: Union[Tuple[Decimal, Decimal], List[Decimal]]*, *point2: Union[Tuple[Decimal, Decimal], List[Decimal]]*) → Decimal
pydash.numerical.**slope**(*point1: Union[Tuple[Union[float, int], Union[float, int]], List[Union[int, float]]]*, *point2: Union[Tuple[Union[float, int], Union[float, int]], List[Union[int, float]]]*) → float

Calculate the slope between two points.

Parameters

- **point1** – X and Y coordinates of first point.
- **point2** – X and Y cooredinates of second point.

Returns Calculated slope.

Example

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

New in version 2.1.0.

pydash.numerical.**std_deviation**(*array: List[Union[float, int, Decimal]]*) → float
Calculate standard deviation of list of numbers.

Parameters **array** – List to process.

Returns Calculated standard deviation.

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.**subtract**(*minuend: SupportsSub[T, T2]*, *subtrahend: pydash.numerical.T*) → pydash.numerical.T2
pydash.numerical.**subtract**(*minuend: pydash.numerical.T*, *subtrahend: SupportsSub[T, T2]*) → pydash.numerical.T2
Subtracts two numbers.

Parameters

- **minuend** – Value passed in by the user.
- **subtrahend** – Value passed in by the user.

Returns Result of the difference from the given values.

Example

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

New in version 4.0.0.

pydash.numerical.**sum_**(*collection: Mapping[Any, SupportsAdd[int, T]]*) → pydash.numerical.T
pydash.numerical.**sum_**(*collection: Iterable[SupportsAdd[int, T]]*) → pydash.numerical.T
Sum each element in *collection*.

Parameters **collection** – Collection to process or first number to add.

Returns Result of summation.

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: Mapping[pydash.numerical.T, pydash.numerical.T2]*, *iteratee: Callable[[pydash.numerical.T2, pydash.numerical.T, Dict[pydash.numerical.T, pydash.numerical.T2]], SupportsAdd[int, T3]]*) → pydash.numerical.T3
pydash.numerical.**sum_by**(*collection: Mapping[pydash.numerical.T, pydash.numerical.T2]*, *iteratee: Callable[[pydash.numerical.T2, pydash.numerical.T], SupportsAdd[int, T3]]*) → pydash.numerical.T3
pydash.numerical.**sum_by**(*collection: Mapping[Any, pydash.numerical.T2]*, *iteratee: Callable[[pydash.numerical.T2], SupportsAdd[int, T3]]*) → pydash.numerical.T3
pydash.numerical.**sum_by**(*collection: Iterable[pydash.numerical.T]*, *iteratee: Callable[[pydash.numerical.T, int, List[pydash.numerical.T]], SupportsAdd[int, T2]]*) → pydash.numerical.T2
pydash.numerical.**sum_by**(*collection: Iterable[pydash.numerical.T]*, *iteratee: Callable[[pydash.numerical.T, int], SupportsAdd[int, T2]]*) → pydash.numerical.T2
pydash.numerical.**sum_by**(*collection: Iterable[pydash.numerical.T]*, *iteratee: Callable[[pydash.numerical.T], SupportsAdd[int, T2]]*) → pydash.numerical.T2
pydash.numerical.**sum_by**(*collection: Mapping[Any, SupportsAdd[int, T]]*, *iteratee: None = None*) → pydash.numerical.T
pydash.numerical.**sum_by**(*collection: Iterable[SupportsAdd[int, T]]*, *iteratee: None = None*) → pydash.numerical.T

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

Parameters

- **collection** – Collection to process or first number to add.
- **iteratee** – Iteratee applied per iteration or second number to add.

Returns Result of summation.

Example

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

New in version 4.0.0.

pydash.numerical.**transpose**(*array: Iterable[Iterable[pydash.numerical.T]]*) → List[List[pydash.numerical.T]]

Transpose the elements of *array*.

Parameters **array** – List to process.

Returns Transposed 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: Mapping[Any, SupportsAdd[int, t.Any]]*) → float
pydash.numerical.**variance**(*array: Iterable[SupportsAdd[int, t.Any]]*) → float
Calculate the variance of the elements in *array*.

Parameters **array** – List to process.

Returns Calculated variance.

Example

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

New in version 2.1.0.

pydash.numerical.**zscore**(*collection: Mapping[pydash.numerical.T, pydash.numerical.T2], iteratee: Callable[[pydash.numerical.T2, pydash.numerical.T, Dict[pydash.numerical.T, pydash.numerical.T2]], Union[float, int, Decimal]]*) → List[float]
pydash.numerical.**zscore**(*collection: Mapping[pydash.numerical.T, pydash.numerical.T2], iteratee: Callable[[pydash.numerical.T2, pydash.numerical.T], Union[float, int, Decimal]]*) → List[float]
pydash.numerical.**zscore**(*collection: Mapping[Any, pydash.numerical.T2], iteratee: Callable[[pydash.numerical.T2], Union[float, int, Decimal]]*) → List[float]
pydash.numerical.**zscore**(*collection: Iterable[pydash.numerical.T], iteratee: Callable[[pydash.numerical.T, int, List[pydash.numerical.T]], Union[float, int, Decimal]]*) → List[float]
pydash.numerical.**zscore**(*collection: Iterable[pydash.numerical.T], iteratee: Callable[[pydash.numerical.T, int], Union[float, int, Decimal]]*) → List[float]
pydash.numerical.**zscore**(*collection: Iterable[pydash.numerical.T], iteratee: Callable[[pydash.numerical.T], Union[float, int, Decimal]]*) → List[float]
pydash.numerical.**zscore**(*collection: Iterable[Union[float, int, Decimal]], iteratee: None = None*) → List[float]
Calculate the standard score assuming normal distribution. If iteratee is passed, each element of *collection* is passed through an iteratee before the standard score is computed.

Parameters

- **collection** – Collection to process.
- **iteratee** – Iteratee applied per iteration.

Returns Calculated standard score.

Example

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

[-1.224744871391589, 0.0, 1.224744871391589]

New in version 2.1.0.

5.1.7 Objects

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

New in version 1.0.0.

pydash.objects.**assign**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, **sources: Mapping[pydash.objects.T3, pydash.objects.T4]*) → Dict[Union[pydash.objects.T, pydash.objects.T3], Union[pydash.objects.T2, pydash.objects.T4]]
pydash.objects.**assign**(*obj: Union[Tuple[pydash.objects.T, ...], List[pydash.objects.T]]*, **sources: Mapping[int, pydash.objects.T2]*) → List[Union[pydash.objects.T, pydash.objects.T2]]

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

Parameters

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

Returns Modified *obj*.

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: Mapping[pydash.objects.T, pydash.objects.T2]*, **sources: Mapping[pydash.objects.T3, Any]*, *customizer: Callable[[Optional[pydash.objects.T2]], pydash.objects.T5]*) → Dict[Union[pydash.objects.T, pydash.objects.T3], Union[pydash.objects.T2, pydash.objects.T5]]
pydash.objects.**assign_with**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, **sources: Mapping[pydash.objects.T3, pydash.objects.T4]*, *customizer: Callable[[Optional[pydash.objects.T2], pydash.objects.T4], pydash.objects.T5]*) → Dict[Union[pydash.objects.T, pydash.objects.T3], Union[pydash.objects.T2, pydash.objects.T5]]
pydash.objects.**assign_with**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, **sources: Mapping[pydash.objects.T3, pydash.objects.T4]*, *customizer: Callable[[Optional[pydash.objects.T2], pydash.objects.T4, pydash.objects.T3], pydash.objects.T5]*) → Dict[Union[pydash.objects.T, pydash.objects.T3], Union[pydash.objects.T2, pydash.objects.T5]]
pydash.objects.**assign_with**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, **sources: Mapping[pydash.objects.T3, pydash.objects.T4]*, *customizer: Callable[[Optional[pydash.objects.T2], pydash.objects.T4, pydash.objects.T3, Dict[pydash.objects.T, pydash.objects.T2]], pydash.objects.T5]*) → Dict[Union[pydash.objects.T, pydash.objects.T3], Union[pydash.objects.T2, pydash.objects.T5]]
pydash.objects.**assign_with**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, **sources: Mapping[pydash.objects.T3, pydash.objects.T4]*, *customizer: Callable[[Optional[pydash.objects.T2], pydash.objects.T4, pydash.objects.T3, Dict[pydash.objects.T, pydash.objects.T2], Dict[pydash.objects.T3, pydash.objects.T4]], pydash.objects.T5]*) → Dict[Union[pydash.objects.T, pydash.objects.T3], Union[pydash.objects.T2, pydash.objects.T5]]
pydash.objects.**assign_with**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, **sources: Mapping[pydash.objects.T3, pydash.objects.T4]*, *customizer: None = 'None'*) → Dict[Union[pydash.objects.T, pydash.objects.T3], Union[pydash.objects.T2, pydash.objects.T4]]

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

Keyword Arguments **customizer** – Customizer applied per iteration.

Returns Modified *obj*.

Warning: *obj* is modified in place.

Example

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

New in version 4.0.0.

pydash.objects.**callables**(*obj: Mapping[SupportsRichComparisonT, Any]*) → List[SupportsRichComparisonT]
pydash.objects.**callables**(*obj: Iterable[pydash.objects.T]*) → List[pydash.objects.T]
Creates a sorted list of keys of an object that are callable.

Parameters **obj** – Object to inspect.

Returns All keys whose values are callable.

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: pydash.objects.T*) → pydash.objects.T
Creates a clone of *value*.

Parameters **value** – 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.

New in version 1.0.0.

Changed in version 4.0.0: Moved 'iteratee' parameter to *clone_with()*.

pydash.objects.**clone_deep**(*value: pydash.objects.T*) → pydash.objects.T
Creates a deep clone of *value*. If an iteratee is provided it will be executed to produce the cloned values.

Parameters **value** – Object to clone.

Returns Cloned object.

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: Mapping[pydash.objects.T, pydash.objects.T2], customizer: Callable[[pydash.objects.T2, pydash.objects.T, Mapping[pydash.objects.T, pydash.objects.T2]], pydash.objects.T3]*) → Dict[pydash.objects.T, Union[pydash.objects.T2, pydash.objects.T3]]
pydash.objects.**clone_deep_with**(*value: Mapping[pydash.objects.T, pydash.objects.T2], customizer: Callable[[pydash.objects.T2, pydash.objects.T], pydash.objects.T3]*) → Dict[pydash.objects.T, Union[pydash.objects.T2, pydash.objects.T3]]
pydash.objects.**clone_deep_with**(*value: Mapping[pydash.objects.T, pydash.objects.T2], customizer: Callable[[pydash.objects.T2], pydash.objects.T3]*) → Dict[pydash.objects.T, Union[pydash.objects.T2, pydash.objects.T3]]
pydash.objects.**clone_deep_with**(*value: List[pydash.objects.T], customizer: Callable[[pydash.objects.T, int, List[pydash.objects.T]], pydash.objects.T2]*) → List[Union[pydash.objects.T, pydash.objects.T2]]
pydash.objects.**clone_deep_with**(*value: List[pydash.objects.T], customizer: Callable[[pydash.objects.T, int], pydash.objects.T2]*) → List[Union[pydash.objects.T, pydash.objects.T2]]
pydash.objects.**clone_deep_with**(*value: List[pydash.objects.T], customizer: Callable[[pydash.objects.T], pydash.objects.T2]*) → List[Union[pydash.objects.T, pydash.objects.T2]]
pydash.objects.**clone_deep_with**(*value: pydash.objects.T, customizer: None = None*) → pydash.objects.T
pydash.objects.**clone_deep_with**(*value: Any, customizer: Callable*) → Any

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

Parameters

- **value** – Object to clone.
- **customizer** – Function to customize cloning.

Returns Cloned object.

pydash.objects.**clone_with**(*value: Mapping[pydash.objects.T, pydash.objects.T2], customizer: Callable[[pydash.objects.T2, pydash.objects.T, Mapping[pydash.objects.T, pydash.objects.T2]], pydash.objects.T3]*) → Dict[pydash.objects.T, Union[pydash.objects.T2, pydash.objects.T3]]
pydash.objects.**clone_with**(*value: Mapping[pydash.objects.T, pydash.objects.T2], customizer: Callable[[pydash.objects.T2, pydash.objects.T], pydash.objects.T3]*) → Dict[pydash.objects.T, Union[pydash.objects.T2, pydash.objects.T3]]
pydash.objects.**clone_with**(*value: Mapping[pydash.objects.T, pydash.objects.T2], customizer: Callable[[pydash.objects.T2], pydash.objects.T3]*) → Dict[pydash.objects.T, Union[pydash.objects.T2, pydash.objects.T3]]
pydash.objects.**clone_with**(*value: List[pydash.objects.T], customizer: Callable[[pydash.objects.T, int, List[pydash.objects.T]], pydash.objects.T2]*) → List[Union[pydash.objects.T, pydash.objects.T2]]

pydash.objects.**clone_with**(*value: List[pydash.objects.T]*, *customizer: Callable[[pydash.objects.T, int], pydash.objects.T2]*) → List[Union[pydash.objects.T, pydash.objects.T2]]
pydash.objects.**clone_with**(*value: List[pydash.objects.T]*, *customizer: Callable[[pydash.objects.T], pydash.objects.T2]*) → List[Union[pydash.objects.T, pydash.objects.T2]]
pydash.objects.**clone_with**(*value: pydash.objects.T*, *customizer: None = None*) → pydash.objects.T
pydash.objects.**clone_with**(*value: Any*, *customizer: Callable*) → Any

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** – Object to clone.
- **customizer** – Function to customize cloning.

Returns Cloned object.

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: Dict[pydash.objects.T, pydash.objects.T2]*, **sources: Dict[pydash.objects.T3, pydash.objects.T4]*) → Dict[Union[pydash.objects.T, pydash.objects.T3], Union[pydash.objects.T2, pydash.objects.T4]]

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

Parameters

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

Returns Modified *obj*.

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: Dict[pydash.objects.T, pydash.objects.T2]*, **sources: Dict[pydash.objects.T3, pydash.objects.T4]*) → Dict[Union[pydash.objects.T, pydash.objects.T3], Union[pydash.objects.T2, pydash.objects.T4]]

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

Parameters

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

Returns Modified *obj*.

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: Mapping[pydash.objects.T, pydash.objects.T2]*, *predicate: Callable[[pydash.objects.T2, pydash.objects.T, Dict[pydash.objects.T, pydash.objects.T2]], Any]*) → Optional[pydash.objects.T]
pydash.objects.**find_key**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, *predicate: Callable[[pydash.objects.T2, pydash.objects.T], Any]*) → Optional[pydash.objects.T]
pydash.objects.**find_key**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, *predicate: Callable[[pydash.objects.T2], Any]*) → Optional[pydash.objects.T]
pydash.objects.**find_key**(*obj: Mapping[pydash.objects.T, Any]*, *predicate: None = None*) → Optional[pydash.objects.T]
pydash.objects.**find_key**(*collection: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T, int, List[pydash.objects.T]], Any]*) → Optional[int]
pydash.objects.**find_key**(*collection: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T, int], Any]*) → Optional[int]
pydash.objects.**find_key**(*collection: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T], Any]*) → Optional[int]
pydash.objects.**find_key**(*collection: Iterable[Any]*, *iteratee: None = None*) → Optional[int]

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** – Object to search.
- **predicate** – Predicate applied per iteration.

Returns Found key or `None`.

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: Mapping[pydash.objects.T, pydash.objects.T2], predicate: Callable[[pydash.objects.T2, pydash.objects.T, Dict[pydash.objects.T, pydash.objects.T2]], Any]*) → Optional[pydash.objects.T]
pydash.objects.**find_last_key**(*obj: Mapping[pydash.objects.T, pydash.objects.T2], predicate: Callable[[pydash.objects.T2, pydash.objects.T], Any]*) → Optional[pydash.objects.T]
pydash.objects.**find_last_key**(*obj: Mapping[pydash.objects.T, pydash.objects.T2], predicate: Callable[[pydash.objects.T2], Any]*) → Optional[pydash.objects.T]
pydash.objects.**find_last_key**(*obj: Mapping[pydash.objects.T, Any], predicate: None = None*) → Optional[pydash.objects.T]
pydash.objects.**find_last_key**(*collection: Iterable[pydash.objects.T], iteratee: Callable[[pydash.objects.T, int, List[pydash.objects.T]], Any]*) → Optional[int]
pydash.objects.**find_last_key**(*collection: Iterable[pydash.objects.T], iteratee: Callable[[pydash.objects.T, int], Any]*) → Optional[int]
pydash.objects.**find_last_key**(*collection: Iterable[pydash.objects.T], iteratee: Callable[[pydash.objects.T], Any]*) → Optional[int]
pydash.objects.**find_last_key**(*collection: Iterable[Any], iteratee: None = None*) → Optional[int]

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

Parameters

- **obj** – Object to search.
- **predicate** – Predicate applied per iteration.

Returns Found key or `None`.

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: Mapping[pydash.objects.T, pydash.objects.T2], iteratee: Callable[[pydash.objects.T2, pydash.objects.T, Dict[pydash.objects.T, pydash.objects.T2]], Any]*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**for_in**(*obj: Mapping[pydash.objects.T, pydash.objects.T2], iteratee: Callable[[pydash.objects.T2, pydash.objects.T], Any]*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**for_in**(*obj: Mapping[pydash.objects.T, pydash.objects.T2], iteratee: Callable[[pydash.objects.T2], Any]*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**for_in**(*obj: Mapping[pydash.objects.T, pydash.objects.T2], iteratee: None = None*) → Dict[pydash.objects.T, pydash.objects.T2]

pydash.objects.**for_in**(*obj: Sequence[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T, int, List[pydash.objects.T]], Any]*) → List[pydash.objects.T]
pydash.objects.**for_in**(*obj: Sequence[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T, int], Any]*) → List[pydash.objects.T]
pydash.objects.**for_in**(*obj: Sequence[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T], Any]*) → List[pydash.objects.T]
pydash.objects.**for_in**(*obj: Sequence[pydash.objects.T]*, *iteratee: None = None*) → List[pydash.objects.T]

Iterates over own and inherited enumerable properties of *obj*, executing *iteratee* for each property.

Parameters

- **obj** – Object to process.
- **iteratee** – Iteratee applied per iteration.

Returns *obj*.

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: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T2, pydash.objects.T, Dict[pydash.objects.T, pydash.objects.T2]], Any]*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**for_in_right**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T2, pydash.objects.T], Any]*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**for_in_right**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T2], Any]*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**for_in_right**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: None = None*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**for_in_right**(*obj: Sequence[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T, int, List[pydash.objects.T]], Any]*) → List[pydash.objects.T]
pydash.objects.**for_in_right**(*obj: Sequence[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T, int], Any]*) → List[pydash.objects.T]
pydash.objects.**for_in_right**(*obj: Sequence[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T], Any]*) → List[pydash.objects.T]
pydash.objects.**for_in_right**(*obj: Sequence[pydash.objects.T]*, *iteratee: None = None*) → List[pydash.objects.T]

This function is like *for_in()* except it iterates over the properties in reverse order.

Parameters

- **obj** – Object to process.
- **iteratee** – Iteratee applied per iteration.

Returns *obj*.

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: List[pydash.objects.T]*, *path: int*, *default: pydash.objects.T2*) → Union[pydash.objects.T, pydash.objects.T2]
pydash.objects.**get**(*obj: List[pydash.objects.T]*, *path: int*, *default: None = None*) → Optional[pydash.objects.T]
pydash.objects.**get**(*obj: Any*, *path: Union[Hashable, List[Hashable]]*, *default: Any = None*) → Any

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** – Object to process.
- **path** – List or `.` delimited string of path describing path.
- **default** – Default value to return if path doesn't exist. Defaults to `None`.

Returns Value of *obj* at path.

Example

```
>>> get({}, 'a.b.c') is None
True
>>> get({'a': {'b': {'c': [1, 2, 3, 4]}}}, 'a.b.c[1]')
2
>>> 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', 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`.

Changed in version 4.7.6: Fixed bug where getattr is used on Mappings and Sequence in Python 3.5+

`pydash.objects.has`(*obj: Any*, *path: Union[Hashable, List[Hashable]]*) → bool

Checks if *path* exists as a key of *obj*.

Parameters

- **obj** – Object to test.
- **path** – 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*.

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: Mapping[pydash.objects.T, pydash.objects.T2]*) → Dict[pydash.objects.T2, pydash.objects.T]

`pydash.objects.invert`(*obj: Iterable[pydash.objects.T]*) → Dict[pydash.objects.T, int]

Creates an object composed of the inverted keys and values of the given object.

Parameters **obj** – Dict to invert.

Returns Inverted 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: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T2], pydash.objects.T3]*) → Dict[pydash.objects.T3, List[pydash.objects.T]]
pydash.objects.**invert_by**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: None = None*) → Dict[pydash.objects.T2, List[pydash.objects.T]]
pydash.objects.**invert_by**(*obj: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T], pydash.objects.T2]*) → Dict[pydash.objects.T2, List[int]]
pydash.objects.**invert_by**(*obj: Iterable[pydash.objects.T]*, *iteratee: None = None*) → Dict[pydash.objects.T, List[int]]

This method is like *invert()* except that the inverted object is generated from the results of running each element of object through 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** – Object to invert.
- **iteratee** – Iteratee applied per iteration.

Returns Inverted 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: Any*, *path: Union[Hashable, List[Hashable]]*, **args: Any*, ***kwargs: Any*) → Any
Invokes the method at path of object.

Parameters

- **obj** – The object to query.
- **path** – The path of the method to invoke.
- **args** – Arguments to pass to method call.
- **kwargs** – Keyword arguments to pass to method call.

Returns Result of the invoked method.

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: Iterable[pydash.objects.T]*) → List[pydash.objects.T]
pydash.objects.**keys**(*obj: Any*) → List
Creates a list composed of the keys of *obj*.

Parameters **obj** – Object to extract keys from.

Returns List of keys.

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: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T2, pydash.objects.T, Dict[pydash.objects.T, pydash.objects.T2]], pydash.objects.T3]*) → Dict[pydash.objects.T3, pydash.objects.T2]
pydash.objects.**map_keys**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T2, pydash.objects.T], pydash.objects.T3]*) → Dict[pydash.objects.T3, pydash.objects.T2]
pydash.objects.**map_keys**(*obj: Mapping[Any, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T2], pydash.objects.T3]*) → Dict[pydash.objects.T3, pydash.objects.T2]
pydash.objects.**map_keys**(*obj: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T, int, List[pydash.objects.T]], pydash.objects.T2]*) → Dict[pydash.objects.T2, pydash.objects.T]

pydash.objects.**map_keys**(*obj: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T, int], pydash.objects.T2]*) → Dict[pydash.objects.T2, pydash.objects.T]
pydash.objects.**map_keys**(*obj: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T], pydash.objects.T2]*) → Dict[pydash.objects.T2, pydash.objects.T]
pydash.objects.**map_keys**(*obj: Iterable*, *iteratee: Optional[Union[int, str, List, Tuple, Dict]] = None*) → Dict

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 through iteratee. The iteratee is invoked with three arguments: `(value, key, object)`.

Parameters

- **obj** – Object to map.
- **iteratee** – Iteratee applied per iteration.

Returns Results of running *obj* through *iteratee*.

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: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T2, pydash.objects.T, Dict[pydash.objects.T, pydash.objects.T2]], pydash.objects.T3]*) → Dict[pydash.objects.T, pydash.objects.T3]
pydash.objects.**map_values**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T2, pydash.objects.T], pydash.objects.T3]*) → Dict[pydash.objects.T, pydash.objects.T3]
pydash.objects.**map_values**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T2], pydash.objects.T3]*) → Dict[pydash.objects.T, pydash.objects.T3]
pydash.objects.**map_values**(*obj: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T, int, List[pydash.objects.T]], pydash.objects.T2]*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**map_values**(*obj: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T, int], pydash.objects.T2]*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**map_values**(*obj: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T], pydash.objects.T2]*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**map_values**(*obj: Iterable*, *iteratee: Optional[Union[int, str, List, Tuple, Dict]] = None*) → Dict

Creates an object with the same keys as object and values generated by running each string keyed property of object through iteratee. The iteratee is invoked with three arguments: `(value, key, object)`.

Parameters

- **obj** – Object to map.
- **iteratee** – Iteratee applied per iteration.

Returns Results of running *obj* through *iteratee*.

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: Iterable*, *iteratee: Optional[Callable] = None*, *property_path: Any = <pydash.helpers.Unset object>*) → Any

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** – Object to map.
- **iteratee** – Iteratee applied to each value.
- **property_path** – Path key(s) to access.

Returns The modified object.

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: Mapping[pydash.objects.T, pydash.objects.T2]*, **sources: Mapping[pydash.objects.T3, pydash.objects.T4]*) → Dict[Union[pydash.objects.T, pydash.objects.T3], Union[pydash.objects.T2, pydash.objects.T4]]

pydash.objects.**merge**(*obj: Sequence[pydash.objects.T]*, **sources: Sequence[pydash.objects.T2]*) → List[Union[pydash.objects.T, pydash.objects.T2]]

Recursively merges properties of the source object(s) into the destination object. Subsequent sources will overwrite property assignments of previous sources.

Parameters

- **obj** – Destination object to merge source(s) into.
- **sources** – Source objects to merge from. subsequent sources overwrite previous ones.

Returns Merged object.

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()*.

Changed in version 4.9.3: Fixed regression in v4.8.0 that caused exception when *obj* was `None`.

pydash.objects.**merge_with**(*obj: Any*, **sources: Any*, ***kwargs: Any*) → Any

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** – Destination object to merge source(s) into.
- **sources** – Source objects to merge from. subsequent sources overwrite previous ones.

Keyword Arguments **iteratee** – Iteratee function to handle merging (must be passed in as keyword argument).

Returns Merged object.

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.

Changed in version 4.9.3: Fixed regression in v4.8.0 that caused exception when *obj* was `None`.

pydash.objects.**omit**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, **properties: Union[Hashable, List[Hashable]]*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**omit**(*obj: Iterable[pydash.objects.T]*, **properties: Union[Hashable, List[Hashable]]*) → Dict[int, pydash.objects.T]
pydash.objects.**omit**(*obj: Any*, **properties: Union[Hashable, List[Hashable]]*) → Dict

The opposite of *pick()*. This method creates an object composed of the property paths of *obj* that are not omitted.

Parameters

- **obj** – Object to process.
- ***properties** – Property values to omit.

Returns Results of omitting properties.

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
>>> omit({'a': {'b': {'c': 'd'}}}, 'a.b.c') == {'a': {'b': {}}}
True
```

New in version 1.0.0.

Changed in version 4.0.0: Moved iteratee argument to *omit_by()*.

Changed in version 4.2.0: Support deep paths.

pydash.objects.**omit_by**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T2, pydash.objects.T], Any]*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**omit_by**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T2], Any]*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**omit_by**(*obj: Dict[pydash.objects.T, pydash.objects.T2]*, *iteratee: None = None*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**omit_by**(*obj: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T, int], Any]*) → Dict[int, pydash.objects.T]
pydash.objects.**omit_by**(*obj: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T], Any]*) → Dict[int, pydash.objects.T]
pydash.objects.**omit_by**(*obj: List[pydash.objects.T]*, *iteratee: None = None*) → Dict[int, pydash.objects.T]
pydash.objects.**omit_by**(*obj: Any*, *iteratee: Optional[Callable] = None*) → Dict

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** – Object to process.
- **iteratee** – Iteratee used to determine which properties to omit.

Returns Results of omitting properties.

Example

```
>>> omit_by({'a': 1, 'b': '2', 'c': 3}, lambda v: isinstance(v, int))
{'b': '2'}
```

New in version 4.0.0.

Changed in version 4.2.0: Support deep paths for *iteratee*.

pydash.objects.**parse_int**(*value: Any*, *radix: Optional[int] = None*) → Optional[int]

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** – Value to parse.
- **radix** – Base to convert to.

Returns Integer if parsable else `None`.

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: Mapping[pydash.objects.T, pydash.objects.T2]*, **properties: Union[Hashable, List[Hashable]]*) → Dict[pydash.objects.T, pydash.objects.T2]

pydash.objects.**pick**(*obj: Union[Tuple[pydash.objects.T, ...], List[pydash.objects.T]]*, **properties: Union[Hashable, List[Hashable]]*) → Dict[int, pydash.objects.T]

pydash.objects.**pick**(*obj: Any*, **properties: Union[Hashable, List[Hashable]]*) → Dict

Creates an object composed of the picked object properties.

Parameters

- **obj** – Object to pick from.
- **properties** – Property values to pick.

Returns Dict containing picked properties.

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: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T2], Any]*) → Dict[pydash.objects.T, pydash.objects.T2]

pydash.objects.**pick_by**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T2, pydash.objects.T], Any]*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**pick_by**(*obj: Dict[pydash.objects.T, pydash.objects.T2]*, *iteratee: None = None*) → Dict[pydash.objects.T, pydash.objects.T2]
pydash.objects.**pick_by**(*obj: Union[Tuple[pydash.objects.T, ...], List[pydash.objects.T]]*, *iteratee: Callable[[pydash.objects.T, int], Any]*) → Dict[int, pydash.objects.T]
pydash.objects.**pick_by**(*obj: Union[Tuple[pydash.objects.T, ...], List[pydash.objects.T]]*, *iteratee: Callable[[pydash.objects.T], Any]*) → Dict[int, pydash.objects.T]
pydash.objects.**pick_by**(*obj: Union[Tuple[pydash.objects.T, ...], List[pydash.objects.T]]*, *iteratee: None = None*) → Dict[int, pydash.objects.T]
pydash.objects.**pick_by**(*obj: Any*, *iteratee: Optional[Callable] = None*) → Dict

Creates an object composed of the object properties predicate returns truthy for. The predicate is invoked with two arguments: `(value, key)`.

Parameters

- **obj** – Object to pick from.
- **iteratee** – Iteratee used to determine which properties to pick.

Returns Dict containing picked properties.

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: Dict[pydash.objects.T, pydash.objects.T2]*, *key_map: Dict[Any, pydash.objects.T3]*) → Dict[Union[pydash.objects.T, pydash.objects.T3], pydash.objects.T2]

Rename the keys of *obj* using *key_map* and return new object.

Parameters

- **obj** – Object to rename.
- **key_map** – Renaming map whose keys correspond to existing keys in *obj* and whose values are the new key name.

Returns Renamed *obj*.

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: pydash.objects.T*, *path: Union[Hashable, List[Hashable]]*, *value: Any*) → pydash.objects.T

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** – Object to modify.
- **path** – Target path to set value to.
- **value** – Value to set.

Returns Modified *obj*.

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: pydash.objects.T*, *path: Union[Hashable, List[Hashable]]*, *value: Any*, *customizer: Optional[Callable] = None*) → pydash.objects.T

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** – Object to modify.
- **path** – Target path to set value to.
- **value** – Value to set.
- **customizer** – The function to customize assigned values.

Returns Modified *obj*.

Warning: *obj* is modified in place.

Example

```
>>> set_with({}, '[0][1]', 'a', lambda: {})
{0: {1: 'a'}}
```

New in version 4.0.0.

Changed in version 4.3.1: Fixed bug where a callable *value* was called when being set.

pydash.objects.**to_boolean**(*obj: Any*, *true_values: Tuple[str, ...] = ('true', '1')*, *false_values: Tuple[str, ...] = ('false', '0')*) → Optional[bool]

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** – Object to convert.
- **true_values** – Values to consider `True`. Each value must be a string. Comparision is case-insensitive. Defaults to `('true', '1')`.
- **false_values** – Values to consider `False`. Each value must be a string. Comparision is case-insensitive. Defaults to `('false', '0')`.

Returns Boolean value of *obj*.

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: Mapping[pydash.objects.T, pydash.objects.T2]*) → Dict[pydash.objects.T, pydash.objects.T2]

pydash.objects.**to_dict**(*obj: Iterable[pydash.objects.T]*) → Dict[int, pydash.objects.T]

pydash.objects.**to_dict**(*obj: Any*) → Dict

Convert *obj* to `dict` by creating a new `dict` using *obj* keys and values.

Parameters **obj** – Object to convert.

Returns Object converted to `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`.

Changed in version 4.2.0: Use `pydash.helpers.iterator` to generate key/value pairs.

Changed in version 4.7.1: Try to convert to `dict` using `dict()` first, then fallback to using `pydash.helpers.iterator`.

pydash.objects.**to_integer**(*obj: Any*) → int

Converts *obj* to an integer.

Parameters **obj** – Object to convert.

Returns Converted integer or `0` if it can't be converted.

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_list**(*obj: Dict[Any, pydash.objects.T]*, *split_strings: bool = True*) → List[pydash.objects.T]
pydash.objects.**to_list**(*obj: Iterable[pydash.objects.T]*, *split_strings: bool = True*) → List[pydash.objects.T]
pydash.objects.**to_list**(*obj: pydash.objects.T*, *split_strings: bool = True*) → List[pydash.objects.T]

Converts an obj, an iterable or a single item to a list.

Parameters

- **obj** – Object to convert item or wrap.
- **split_strings** – Whether to split strings into single chars. Defaults to `True`.

Returns Converted obj or wrapped item.

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]
```

```
>>> to_list(1)
[1]
```

```
>>> to_list([1])
[1]
```

```
>>> to_list(a for a in [1, 2, 3])
[1, 2, 3]
```

```
>>> to_list('cat')
['c', 'a', 't']
```

```
>>> to_list('cat', split_strings=False)
['cat']
```

New in version 1.0.0.

Changed in version 4.3.0:

- Wrap non-iterable items in a list.
- Convert other iterables to list.
- Byte objects are returned as single character strings in Python 3.

pydash.objects.**to_number**(*obj: Any*, *precision: int = 0*) → Optional[float]

Convert *obj* to a number. All numbers are retuned 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** – Object to convert.
- **precision** – Precision to round number to. Defaults to `0`.

Returns Converted number or `None` if it can't be converted.

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: Mapping[pydash.objects.T, pydash.objects.T2]*) → List[List[Union[pydash.objects.T, pydash.objects.T2]]]
pydash.objects.**to_pairs**(*obj: Iterable[pydash.objects.T]*) → List[List[Union[int, pydash.objects.T]]]
pydash.objects.**to_pairs**(*obj: Any*) → List

Creates a two-dimensional list of an object's key-value pairs, i.e., `[[key1, value1], [key2, value2]]`.

Parameters **obj** – Object to process.

Returns Two dimensional list of object's key-value pairs.

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: Any*) → str

Converts an object to string.

Parameters **obj** – Object to convert.

Returns String representation of *obj*.

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: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T3, pydash.objects.T2, pydash.objects.T, Dict[pydash.objects.T, pydash.objects.T2]], Any]*, *accumulator: pydash.objects.T3*) → pydash.objects.T3
pydash.objects.**transform**(*obj: Mapping[pydash.objects.T, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T3, pydash.objects.T2, pydash.objects.T], Any]*, *accumulator: pydash.objects.T3*) → pydash.objects.T3
pydash.objects.**transform**(*obj: Mapping[Any, pydash.objects.T2]*, *iteratee: Callable[[pydash.objects.T3, pydash.objects.T2], Any]*, *accumulator: pydash.objects.T3*) → pydash.objects.T3
pydash.objects.**transform**(*obj: Mapping[Any, Any]*, *iteratee: Callable[[pydash.objects.T3], Any]*, *accumulator: pydash.objects.T3*) → pydash.objects.T3

pydash.objects.**transform**(*obj: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T3, pydash.objects.T, int, List[pydash.objects.T]], Any]*, *accumulator: pydash.objects.T3*) → pydash.objects.T3
pydash.objects.**transform**(*obj: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T3, pydash.objects.T, int], Any]*, *accumulator: pydash.objects.T3*) → pydash.objects.T3
pydash.objects.**transform**(*obj: Iterable[pydash.objects.T]*, *iteratee: Callable[[pydash.objects.T3, pydash.objects.T], Any]*, *accumulator: pydash.objects.T3*) → pydash.objects.T3
pydash.objects.**transform**(*obj: Iterable[Any]*, *iteratee: Callable[[pydash.objects.T3], Any]*, *accumulator: pydash.objects.T3*) → pydash.objects.T3
pydash.objects.**transform**(*obj: Any*, *iteratee: Any = None*, *accumulator: Any = None*) → Any

An alernative to `pydash.collections.reduce()`, this method transforms *obj* to a new accumulator object which is the result of running each of its properties through an 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** – Object to process.
- **iteratee** – Iteratee applied per iteration.
- **accumulator** – Accumulated object. Defaults to `list`.

Returns Accumulated object.

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: Union[List, Dict]*, *path: Union[Hashable, List[Hashable]]*) → bool

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** – The object to modify.
- **path** – The path of the property to unset.

Returns Whether the property was deleted.

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: Dict[Any, pydash.objects.T2]*, *path: Union[Hashable, List[Hashable]]*, *updater: Callable[[pydash.objects.T2], Any]*) → Dict
pydash.objects.**update**(*obj: List[pydash.objects.T]*, *path: Union[Hashable, List[Hashable]]*, *updater: Callable[[pydash.objects.T], Any]*) → List
pydash.objects.**update**(*obj: pydash.objects.T*, *path: Union[Hashable, List[Hashable]]*, *updater: Callable*) → pydash.objects.T

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** – Object to modify.
- **path** – A string or list of keys that describe the object path to modify.
- **updater** – Function that returns updated value.

Returns Updated *obj*.

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: Dict[Any, pydash.objects.T2]*, *path: Union[Hashable, List[Hashable]]*, *updater: Callable[[pydash.objects.T2], Any]*, *customizer: Optional[Callable]*) → Dict
pydash.objects.**update_with**(*obj: List[pydash.objects.T]*, *path: Union[Hashable, List[Hashable]]*, *updater: Callable[[pydash.objects.T], Any]*, *customizer: Optional[Callable] = None*) → List
pydash.objects.**update_with**(*obj: pydash.objects.T*, *path: Union[Hashable, List[Hashable]]*, *updater: Callable*, *customizer: Optional[Callable] = None*) → pydash.objects.T

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** – Object to modify.

- **path** – A string or list of keys that describe the object path to modify.
- **updater** – Function that returns updated value.
- **customizer** – The function to customize assigned values.

Returns Updated *obj*.

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: Mapping[Any, pydash.objects.T2]*) → List[pydash.objects.T2]
pydash.objects.**values**(*obj: Iterable[pydash.objects.T]*) → List[pydash.objects.T]
pydash.objects.**values**(*obj: Any*) → List

Creates a list composed of the values of *obj*.

Parameters **obj** – Object to extract values from.

Returns List of values.

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`.

5.1.8 Predicates

Predicate functions that return boolean evaluations of objects.

New in version 2.0.0.

pydash.predicates.**eq**(*value: Any*, *other: Any*) → bool

Checks if `value` is equal to `other`.

Parameters

- **value** – Value to compare.
- **other** – Other value to compare.

Returns Whether `value` is equal to `other`.

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: SupportsDunderGT[T]*, *other: pydash.predicates.T*) → bool

Checks if *value* is greater than *other*.

Parameters

- **value** – Value to compare.
- **other** – Other value to compare.

Returns Whether *value* is greater than *other*.

Example

```
>>> gt(5, 3)
True
>>> gt(3, 5)
False
>>> gt(5, 5)
False
```

New in version 3.3.0.

pydash.predicates.**gte**(*value: SupportsDunderGE[T]*, *other: pydash.predicates.T*) → bool

Checks if *value* is greater than or equal to *other*.

Parameters

- **value** – Value to compare.
- **other** – Other value to compare.

Returns Whether *value* is greater than or equal to *other*.

Example

```
>>> gte(5, 3)
True
>>> gte(3, 5)
False
>>> gte(5, 5)
True
```

New in version 3.3.0.

pydash.predicates.**in_range**(*value: Any*, *start: Any = 0*, *end: Optional[Any] = None*) → bool

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** – Number to check.
- **start** – Start of range inclusive. Defaults to `0`.
- **end** – End of range exclusive. Defaults to *start*.

Returns Whether *value* is in range.

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: Any*) → bool

Checks if *value* is an associative object meaning that it can be accessed via an index or key.

Parameters **value** – Value to check.

Returns Whether *value* is associative.

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: Any*) → typing_extensions.TypeGuard[str]

Checks if *text* contains only whitespace characters.

Parameters **text** – String to test.

Returns Whether *text* is blank.

Example

```
>>> is_blank('')
True
>>> is_blank(' \r\n ')
True
>>> is_blank(False)
False
```

New in version 3.0.0.

pydash.predicates.**is_boolean**(*value: Any*) → typing_extensions.TypeGuard[bool]
Checks if *value* is a boolean value.

Parameters **value** – Value to check.

Returns Whether *value* is a boolean.

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: Any*) → bool
Checks if *value* is a Python builtin function or method.

Parameters **value** – Value to check.

Returns Whether *value* is a Python builtin function or method.

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: Any*) → bool
Check if *value* is a date object.

Parameters **value** – Value to check.

Returns Whether *value* is a date object.

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: Union[SupportsRichComparison, List[SupportsRichComparison]]*) → bool

Check if *value* is monotonically decreasing.

Parameters **value** – Value to check.

Returns Whether *value* is monotonically decreasing.

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: Any*) → bool

Checks if *value* is a `dict`.

Parameters **value** – Value to check.

Returns Whether *value* is a `dict`.

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: Any*) → bool
Checks if *value* is empty.

Parameters **value** – Value to check.

Returns Whether *value* is empty.

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: Any*, *other: Any*) → bool
Performs a comparison between two values to determine if they are equivalent to each other.

Parameters

- **value** – Object to compare.
- **other** – Object to compare.

Returns Whether *value* and *other* are equal.

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: pydash.predicates.T*, *other: pydash.predicates.T2*, *customizer: Callable[[pydash.predicates.T, pydash.predicates.T2], pydash.predicates.T3]*) → pydash.predicates.T3
pydash.predicates.**is_equal_with**(*value: Any*, *other: Any*, *customizer: Callable*) → bool

pydash.predicates.**is_equal_with**(*value: Any*, *other: Any*, *customizer: None*) → bool

This method is like *is_equal()* 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** – Object to compare.
- **other** – Object to compare.
- **customizer** – Customizer used to compare values from *value* and *other*.

Returns Whether *value* and *other* are equal.

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: Any*) → bool

Checks if *value* is an `Exception`.

Parameters **value** – Value to check.

Returns Whether *value* is an exception.

Example

```
>>> is_error(Exception())
True
>>> is_error(Exception)
False
>>> is_error(None)
False
```

New in version 1.1.0.

pydash.predicates.**is_even**(*value: Any*) → bool

Checks if *value* is even.

Parameters **value** – Value to check.

Returns Whether *value* is even.

Example

```
>>> is_even(2)
True
>>> is_even(3)
False
>>> is_even(False)
False
```

New in version 2.0.0.

pydash.predicates.**is_float**(*value: Any*) → typing_extensions.TypeGuard[float]
Checks if *value* is a float.

Parameters **value** – Value to check.

Returns Whether *value* is a float.

Example

```
>>> is_float(1.0)
True
>>> is_float(1)
False
```

New in version 2.0.0.

pydash.predicates.**is_function**(*value: Any*) → bool
Checks if *value* is a function.

Parameters **value** – Value to check.

Returns Whether *value* is callable.

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: Union[SupportsRichComparison, List[SupportsRichComparison]]*) → bool
Check if *value* is monotonically increasing.

Parameters **value** – Value to check.

Returns Whether *value* is monotonically increasing.

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: Any*) → bool
Checks if *value* is integer indexed, i.e., `list`, `str` or `tuple`.

Parameters **value** – Value to check.

Returns Whether *value* is integer indexed.

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: Any*, *types: Union[type, Tuple[type, ...]]*) → bool
Checks if *value* is an instance of *types*.

Parameters

- **value** – Value to check.
- **types** – Types to check against. Pass as `tuple` to check if *value* is one of multiple types.

Returns Whether *value* is an instance of *types*.

Example

```
>>> is_instance_of({}, dict)
True
>>> is_instance_of({}, list)
False
```

New in version 2.0.0.

pydash.predicates.**is_integer**(*value: Any*) → typing_extensions.TypeGuard[int]
Checks if *value* is a integer.

Parameters `value` – Value to check.

Returns Whether *value* is an integer.

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: Any*) → bool

Checks if *value* is an iterable.

Parameters `value` – Value to check.

Returns Whether *value* is an iterable.

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: Any*) → bool

Checks if *value* is a valid JSON string.

Parameters `value` – Value to check.

Returns Whether *value* is JSON.

Example

```
>>> is_json({})
False
>>> is_json('{}')
True
>>> is_json({"hello": 1, "world": 2})
False
>>> is_json('{"hello": 1, "world": 2}')
True
```

New in version 2.0.0.

pydash.predicates.**is_list**(*value: Any*) → bool

Checks if *value* is a list.

Parameters **value** – Value to check.

Returns Whether *value* is a list.

Example

```
>>> is_list([])
True
>>> is_list({})
False
>>> is_list(())
False
```

New in version 1.0.0.

pydash.predicates.**is_match**(*obj: Any*, *source: Any*) → bool

Performs a partial deep comparison between *obj* and *source* to determine if *obj* contains equivalent property values.

Parameters

- **obj** – Object to compare.
- **source** – Object of property values to match.

Returns Whether *obj* is a match or not.

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': [{
↪'d': 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: Any*, *source: Any*, *customizer: Optional[Any] = None*, *_key: Any = <pydash.helpers.Unset object>*, *_obj: Any = <pydash.helpers.Unset object>*, *_source: Any = <pydash.helpers.Unset object>*) → bool

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** – Object to compare.
- **source** – Object of property values to match.
- **customizer** – Customizer used to compare values from *obj* and *source*.

Returns Whether *obj* is a match or not.

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: Union[pydash.predicates.T, List[pydash.predicates.T]]*, *op: Callable[[pydash.predicates.T, pydash.predicates.T], Any]*) → bool

Checks if *value* is monotonic when *operator* used for comparison.

Parameters

- **value** – Value to check.
- **op** – Operation to used for comparison.

Returns Whether *value* is monotone.

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: Any*) → bool

Checks if *value* is not a number.

Parameters **value** – Value to check.

Returns Whether *value* is not a number.

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: Any*) → bool
Checks if *value* is negative.

Parameters **value** – Value to check.

Returns Whether *value* is negative.

Example

```
>>> is_negative(-1)
True
>>> is_negative(0)
False
>>> is_negative(1)
False
```

New in version 2.0.0.

pydash.predicates.**is_none**(*value: Any*) → typing_extensions.TypeGuard[None]
Checks if *value* is *None*.

Parameters **value** – Value to check.

Returns Whether *value* is `None`.

Example

```
>>> is_none(None)
True
>>> is_none(False)
False
```

New in version 1.0.0.

pydash.predicates.**is_number**(*value: Any*) → bool
Checks if *value* is a number.

Parameters **value** – Value to check.

Returns Whether *value* is a number.

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: Any*) → bool
Checks if *value* is a `list` or `dict`.

Parameters **value** – Value to check.

Returns Whether *value* is `list` or `dict`.

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: Any*) → bool
Checks if *value* is odd.

Parameters **value** – Value to check.

Returns Whether *value* is odd.

Example

```
>>> is_odd(3)
True
>>> is_odd(2)
False
>>> is_odd('a')
False
```

New in version 2.0.0.

pydash.predicates.**is_positive**(*value: Any*) → bool
Checks if *value* is positive.

Parameters **value** – Value to check.

Returns Whether *value* is positive.

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: Any*) → typing_extensions.TypeGuard[re.Pattern]
Checks if *value* is a `RegExp` object.

Parameters **value** – Value to check.

Returns Whether *value* is a RegExp object.

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: Any*) → bool
Checks if the given value is a set object or not.

Parameters **value** – Value passed in by the user.

Returns True if the given value is a set else False.

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: Union[SupportsRichComparison, List[SupportsRichComparison]]*) → bool
Check if *value* is strictly decreasing.

Parameters **value** – Value to check.

Returns Whether *value* is strictly decreasing.

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: Union[SupportsRichComparison, List[SupportsRichComparison]]*) → bool

Check if *value* is strictly increasing.

Parameters **value** – Value to check.

Returns Whether *value* is strictly increasing.

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: Any*) → typing_extensions.TypeGuard[str]

Checks if *value* is a string.

Parameters **value** – Value to check.

Returns Whether *value* is a string.

Example

```
>>> is_string('')
True
>>> is_string(1)
False
```

New in version 1.0.0.

pydash.predicates.**is_tuple**(*value: Any*) → bool

Checks if *value* is a tuple.

Parameters **value** – Value to check.

Returns Whether *value* is a tuple.

Example

```
>>> is_tuple(())
True
>>> is_tuple({})
False
>>> is_tuple([])
False
```

New in version 3.0.0.

pydash.predicates.**is_zero**(*value: Any*) → typing_extensions.TypeGuard[int]
Checks if *value* is `0`.

Parameters **value** – Value to check.

Returns Whether *value* is `0`.

Example

```
>>> is_zero(0)
True
>>> is_zero(1)
False
```

New in version 2.0.0.

pydash.predicates.**lt**(*value: SupportsDunderLT[T]*, *other: pydash.predicates.T*) → bool
Checks if *value* is less than *other*.

Parameters

- **value** – Value to compare.
- **other** – Other value to compare.

Returns Whether *value* is less than *other*.

Example

```
>>> lt(5, 3)
False
>>> lt(3, 5)
True
>>> lt(5, 5)
False
```

New in version 3.3.0.

pydash.predicates.**lte**(*value: SupportsDunderLE[T]*, *other: pydash.predicates.T*) → bool
Checks if *value* is less than or equal to *other*.

Parameters

- **value** – Value to compare.
- **other** – Other value to compare.

Returns Whether *value* is less than or equal to *other*.

Example

```
>>> lte(5, 3)
False
>>> lte(3, 5)
True
>>> lte(5, 5)
True
```

New in version 3.3.0.

5.1.9 Strings

String functions.

New in version 1.1.0.

pydash.strings.**camel_case**(*text: Any*) → str
Converts *text* to camel case.

Parameters **text** – String to convert.

Returns String converted to camel case.

Example

```
>>> camel_case('FOO BAR_bAz')
'fooBarBAz'
```

New in version 1.1.0.

Changed in version 5.0.0: Improved unicode word support.

pydash.strings.**capitalize**(*text: Any*, *strict: bool = True*) → str
Capitalizes the first character of *text*.

Parameters

- **text** – String to capitalize.
- **strict** – Whether to cast rest of string to lower case. Defaults to `True`.

Returns Capitalized string.

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.**chars**(*text: Any*) → List[str]
Split *text* into a list of single characters.

Parameters **text** – String to split up.

Returns List of individual characters.

Example

```
>>> chars('onetwo')
['o', 'n', 'e', 't', 'w', 'o']
```

New in version 3.0.0.

pydash.strings.**chop**(*text: Any*, *step: int*) → List[str]
Break up *text* into intervals of length *step*.

Parameters

- **text** – String to chop.
- **step** – Interval to chop *text*.

Returns List of chopped characters. If *text* is *None* an empty list is returned.

Example

```
>>> chop('abcdefg', 3)
['abc', 'def', 'g']
```

New in version 3.0.0.

pydash.strings.**chop_right**(*text: Any*, *step: int*) → List[str]
Like *chop()* except *text* is chopped from right.

Parameters

- **text** – String to chop.
- **step** – Interval to chop *text*.

Returns List of chopped characters.

Example

```
>>> chop_right('abcdefg', 3)
['a', 'bcd', 'efg']
```

New in version 3.0.0.

pydash.strings.**clean**(*text: Any*) → str
Trim and replace multiple spaces with a single space.

Parameters **text** – String to clean.

Returns Cleaned string.

Example

```
>>> clean('a  b   c    d')
'a b c d'
```

New in version 3.0.0.

pydash.strings.**count_substr**(*text: Any*, *subtext: Any*) → int
Count the occurrences of *subtext* in *text*.

Parameters

- **text** – Source string to count from.
- **subtext** – String to count.

Returns Number of occurrences of *subtext* in *text*.

Example

```
>>> count_substr('aabbccddaabbccdd', 'bc')
2
```

New in version 3.0.0.

pydash.strings.**deburr**(*text: Any*) → str
Deburrs *text* by converting latin-1 supplementary letters to basic latin letters.

Parameters **text** – String to deburr.

Returns Deburred string.

Example

```
>>> deburr('déjà vu')
'...
>>> 'deja vu'
'deja vu'
```

New in version 2.0.0.

pydash.strings.**decapitalize**(*text: Any*) → str
Decaptitalizes the first character of *text*.

Parameters **text** – String to decapitalize.

Returns Decapitalized string.

Example

```
>>> decapitalize('FOO BAR')
'fOO BAR'
```

New in version 3.0.0.

pydash.strings.**ends_with**(*text: Any*, *target: Any*, *position: Optional[int] = None*) → bool
Checks if *text* ends with a given target string.

Parameters

- **text** – String to check.
- **target** – String to check for.
- **position** – Position to search from. Defaults to end of *text*.

Returns Whether *text* ends with *target*.

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: Any*, *suffix: Any*) → str
Append a given suffix to a string, but only if the source string does not end with that suffix.

Parameters

- **text** – Source string to append *suffix* to.
- **suffix** – String to append to the source string if the source string does not end with *suffix*.

Returns source string possibly extended by *suffix*.

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: Any*, *prefix: Any*) → str
Prepend a given prefix to a string, but only if the source string does not start with that prefix.

Parameters

- **text** – Source string to prepend *prefix* to.
- **prefix** – String to prepend to the source string if the source string does not start with *prefix*.

Returns source string possibly prefixed by *prefix*

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: Any*) → str
Converts the characters &, <, >, ", ', and \` in *text* to their corresponding HTML entities.

Parameters **text** – String to escape.

Returns HTML escaped string.

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: Any*) → str
Escapes the RegExp special characters in *text*.

Parameters **text** – String to escape.

Returns RegExp escaped string.

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: Any*, *subtext: Any*) → bool
Returns whether *subtext* is included in *text*.

Parameters

- **text** – String to search.
- **subtext** – String to search for.

Returns Whether *subtext* is found in *text*.

Example

```
>>> has_substr('abcdef', 'bc')
True
>>> has_substr('abcdef', 'bb')
False
```

New in version 3.0.0.

pydash.strings.**human_case**(*text: Any*) → str

Converts *text* to human case which has only the first letter capitalized and each word separated by a space.

Parameters **text** – String to convert.

Returns String converted to human case.

Example

```
>>> human_case('abc-def_hij lmn')
'Abc def hij lmn'
>>> human_case('user_id')
'User'
```

New in version 3.0.0.

Changed in version 5.0.0: Improved unicode word support.

pydash.strings.**insert_substr**(*text: Any*, *index: int*, *subtext: Any*) → str

Insert *subtext* in *text* starting at position *index*.

Parameters

- **text** – String to add substring to.
- **index** – String index to insert into.
- **subtext** – String to insert.

Returns Modified string.

Example

```
>>> insert_substr('abcdef', 3, '--')
'abc--def'
```

New in version 3.0.0.

pydash.strings.**join**(*array: Iterable[Any]*, *separator: Any = ''*) → str

Joins an iterable into a string using *separator* between each element.

Parameters

- **array** – Iterable to implode.
- **separator** – Separator to using when joining. Defaults to `''`.

Returns Joined string.

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: Any*) → str
Converts *text* to kebab case (a.k.a. spinal case).

Parameters **text** – String to convert.

Returns String converted to kebab case.

Example

```
>>> kebab_case('a b c_d-e!f')
'a-b-c-d-e-f'
```

New in version 1.1.0.

Changed in version 5.0.0: Improved unicode word support.

pydash.strings.**lines**(*text: Any*) → List[str]
Split lines in *text* into an array.

Parameters **text** – String to split.

Returns String split by lines.

Example

```
>>> lines('a\nb\r\nc')
['a', 'b', 'c']
```

New in version 3.0.0.

pydash.strings.**lower_case**(*text: Any*) → str
Converts string to lower case as space separated words.

Parameters **text** – String to convert.

Returns String converted to lower case as space separated words.

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.

Changed in version 5.0.0: Improved unicode word support.

pydash.strings.**lower_first**(*text: str*) → str

Converts the first character of string to lower case.

Parameters **text** – String passed in by the user.

Returns String in which the first character is converted to lower case.

Example

```
>>> lower_first('FRED')
'fRED'
>>> lower_first('Foo Bar')
'foo Bar'
>>> lower_first('1foobar')
'1foobar'
>>> lower_first(';foobar')
';foobar'
```

New in version 4.0.0.

pydash.strings.**number_format**(*number: Union[float, int, Decimal]*, *scale: int = 0*, *decimal_separator: str = '.'*, *order_separator: str = ','*) → str

Format a number to scale with custom decimal and order separators.

Parameters

- **number** – Number to format.
- **scale** – Number of decimals to include. Defaults to `0`.
- **decimal_separator** – Decimal separator to use. Defaults to `'.'`.
- **order_separator** – Order separator to use. Defaults to `','`.

Returns Number formatted as string.

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: Any*, *length: int*, *chars: Any = ' '*) → str

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** – String to pad.
- **length** – Amount to pad.
- **chars** – Characters to pad with. Defaults to " ".

Returns Padded string.

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: Any*, *length: int*, *chars: Any = ' '*) → str

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** – String to pad.
- **length** – Amount to pad.
- **chars** – Characters to pad with. Defaults to " ".

Returns Padded string.

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: Any*, *length: int*, *chars: Any = ' '*) → str

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** – String to pad.
- **length** – Amount to pad.
- **chars** – Characters to pad with. Defaults to " ".

Returns Padded string.

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: Any*, *strict: bool = True*) → str

Like *camel_case()* except the first letter is capitalized.

Parameters

- **text** – String to convert.
- **strict** – Whether to cast rest of string to lower case. Defaults to `True`.

Returns String converted to class case.

Example

```
>>> pascal_case('FOO BAR_bAz')
'FooBarBaz'
>>> pascal_case('FOO BAR_bAz', False)
'FooBarBAz'
```

New in version 3.0.0.

Changed in version 5.0.0: Improved unicode word support.

pydash.strings.**predecessor**(*char: Any*) → str
Return the predecessor character of *char*.

Parameters **char** – Character to find the predecessor of.

Returns Predecessor character.

Example

```
>>> predecessor('c')
'b'
>>> predecessor('C')
'B'
>>> predecessor('3')
'2'
```

New in version 3.0.0.

pydash.strings.**prune**(*text: Any*, *length: int = 0*, *omission: str = '...'*) → str
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** – String to prune.
- **length** – Target prune length. Defaults to `0`.
- **omission** – Omission text to append to the end of the pruned string. Defaults to `'...'`.

Returns Pruned string.

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: Any*, *quote_char: Any = '"'*) → str
Quote a string with another string.

Parameters

- **text** – String to be quoted.
- **quote_char** – the quote character. Defaults to `'"'`.

Returns the quoted string.

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: Any*, *reg_exp: str*) → List[str]

Return list of matches using Javascript style regular expression.

Parameters

- **text** – String to evaluate.
- **reg_exp** – Javascript style regular expression.

Returns List of matches.

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: Any*, *reg_exp: str*, *repl: Union[str, Callable[[re.Match], str]]*) → str

Replace *text* with *repl* using Javascript style regular expression to find matches.

Parameters

- **text** – String to evaluate.
- **reg_exp** – Javascript style regular expression.
- **repl** – Replacement string or callable.

Returns Modified string.

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: Any*, *pattern: Any*, *repl: Union[str, Callable[[re.Match], str]]*, *ignore_case: bool = False*, *count: int = 0*) → str

Replace occurrences of regex *pattern* with *repl* in *text*. Optionally, ignore case when replacing. Optionally, set *count* to limit number of replacements.

Parameters

- **text** – String to replace.
- **pattern** – Pattern to find and replace.
- **repl** – String to substitute *pattern* with.
- **ignore_case** – Whether to ignore case when replacing. Defaults to `False`.
- **count** – Maximum number of occurrences to replace. Defaults to `0` which replaces all.

Returns Replaced string.

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: Any*, *n: SupportsInt = 0*) → str

Repeats the given string *n* times.

Parameters

- **text** – String to repeat.
- **n** – Number of times to repeat the string.

Returns Repeated string.

Example

```
>>> repeat('.', 5)
'.....'
```

New in version 1.1.0.

pydash.strings.**replace**(*text: Any*, *pattern: Any*, *repl: Union[str, Callable[[re.Match], str]]*, *ignore_case: bool = False*, *count: int = 0*, *escape: bool = True*, *from_start: bool = False*, *from_end: bool = False*) → str

Replace occurrences of *pattern* with *repl* in *text*. Optionally, ignore case when replacing. Optionally, set *count* to limit number of replacements.

Parameters

- **text** – String to replace.
- **pattern** – Pattern to find and replace.
- **repl** – String to substitute *pattern* with.
- **ignore_case** – Whether to ignore case when replacing. Defaults to `False`.
- **count** – Maximum number of occurrences to replace. Defaults to `0` which replaces all.
- **escape** – 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`.
- **from_start** – Whether to limit replacement to start of string.
- **from_end** – Whether to limit replacement to end of string.

Returns Replaced string.

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.

Changed in version 4.1.0: Added `from_start` and `from_end` arguments.

Changed in version 5.0.0: Added support for `pattern` as `typing.Pattern` object.

pydash.strings.**replace_end**(*text: Any*, *pattern: Any*, *repl: Union[str, Callable[[re.Match], str]]*, *ignore_case: bool = False*, *escape: bool = True*) → str

Like *replace()* except it only replaces *text* with *repl* if *pattern* mathces the end of *text*.

Parameters

- **text** – String to replace.

- **pattern** – Pattern to find and replace.
- **repl** – String to substitute *pattern* with.
- **ignore_case** – Whether to ignore case when replacing. Defaults to `False`.
- **escape** – 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.

Example

```
>>> replace_end('aabbcc', 'b', 'X')
'aabbcc'
>>> replace_end('aabbcc', 'c', 'X')
'aabbcX'
```

New in version 4.1.0.

pydash.strings.**replace_start**(*text: Any*, *pattern: Any*, *repl: Union[str, Callable[[re.Match], str]]*, *ignore_case: bool = False*, *escape: bool = True*) → str

Like *replace()* except it only replaces *text* with *repl* if *pattern* mathces the start of *text*.

Parameters

- **text** – String to replace.
- **pattern** – Pattern to find and replace.
- **repl** – String to substitute *pattern* with.
- **ignore_case** – Whether to ignore case when replacing. Defaults to `False`.
- **escape** – 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.

Example

```
>>> replace_start('aabbcc', 'b', 'X')
'aabbcc'
>>> replace_start('aabbcc', 'a', 'X')
'Xabbcc'
```

New in version 4.1.0.

pydash.strings.**separator_case**(*text: Any*, *separator: str*) → str

Splits *text* on words and joins with *separator*.

Parameters

- **text** – String to convert.
- **separator** – Separator to join words with.

Returns Converted string.

Example

```
>>> separator_case('a!!b___c.d', '-')
'a-b-c-d'
```

New in version 3.0.0.

Changed in version 5.0.0: Improved unicode word support.

pydash.strings.**series_phrase**(*items: List[Any]*, *separator: Any = ', '*, *last_separator: Any = ' and '*, *serial: bool = False*) → str

Join items into a grammatical series phrase, e.g., "item1, item2, item3 and item4".

Parameters

- **items** – List of string items to join.
- **separator** – Item separator. Defaults to ', '.
- **last_separator** – Last item separator. Defaults to ' and '.
- **serial** – Whether to include *separator* with *last_separator* when number of items is greater than 2. Defaults to False.

Returns Joined string.

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: List[Any]*, *separator: Any = ', '*, *last_separator: Any = ' and '*) → str

Join items into a grammatical series phrase using a serial separator, e.g., "item1, item2, item3, and item4".

Parameters

- **items** – List of string items to join.
- **separator** – Item separator. Defaults to ', '.
- **last_separator** – Last item separator. Defaults to ' and '.

Returns Joined string.

Example

```
>>> series_phrase_serial(['apples', 'bananas', 'peaches'])
'apples, bananas, and peaches'
```

New in version 3.0.0.

pydash.strings.**slugify**(*text: Any*, *separator: str = '-'*) → str

Convert *text* into an ASCII slug which can be used safely in URLs. Incoming *text* is converted to unicode and noramlzied 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** – String to slugify.
- **separator** – Separator to use. Defaults to `'-'`.

Returns Slugified string.

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.

Changed in version 5.0.0: Improved unicode word support.

Changed in version 7.0.0: Remove single quotes from output.

pydash.strings.**snake_case**(*text: Any*) → str

Converts *text* to snake case.

Parameters **text** – String to convert.

Returns String converted to snake case.

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`.

Changed in version 5.0.0: Improved unicode word support.

pydash.strings.**split**(*text: Any*, *separator: Optional[Union[str, pydash.helpers.Unset]] = <pydash.helpers.Unset object>*) → List[str]

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** – String to explode.
- **separator** – Separator string to split on. Defaults to `NoValue`.

Returns Split string.

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: Any*) → str

Convert *text* to start case.

Parameters **text** – String to convert.

Returns String converted to start case.

Example

```
>>> start_case("fooBar")
'Foo Bar'
```

New in version 3.1.0.

Changed in version 5.0.0: Improved unicode word support.

`pydash.strings.starts_with`(*text: Any*, *target: Any*, *position: int = 0*) → bool

Checks if *text* starts with a given target string.

Parameters

- **text** – String to check.
- **target** – String to check for.
- **position** – Position to search from. Defaults to beginning of *text*.

Returns Whether *text* starts with *target*.

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: Any*) → str
Removes all HTML tags from *text*.

Parameters **text** – String to strip.

Returns String without HTML tags.

Example

```
>>> strip_tags('<a href="#">Some link</a>')
'Some link'
```

New in version 3.0.0.

pydash.strings.**substr_left**(*text: Any*, *subtext: str*) → str
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** – String to partition.
- **subtext** – String to search for.

Returns Substring to left of *subtext*.

Example

```
>>> substr_left('abcdefcdg', 'cd')
'ab'
```

New in version 3.0.0.

pydash.strings.**substr_left_end**(*text: Any*, *subtext: str*) → str
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** – String to partition.
- **subtext** – String to search for.

Returns Substring to left of *subtext*.

Example

```
>>> substr_left_end('abcdefcdg', 'cd')
'abcdef'
```

New in version 3.0.0.

pydash.strings.**substr_right**(*text: Any*, *subtext: str*) → str

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** – String to partition.
- **subtext** – String to search for.

Returns Substring to right of *subtext*.

Example

```
>>> substr_right('abcdefcdg', 'cd')
'efcdg'
```

New in version 3.0.0.

pydash.strings.**substr_right_end**(*text: Any*, *subtext: str*) → str

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** – String to partition.
- **subtext** – String to search for.

Returns Substring to right of *subtext*.

Example

```
>>> substr_right_end('abcdefcdg', 'cd')
'g'
```

New in version 3.0.0.

pydash.strings.**successor**(*char: Any*) → str

Return the successor character of *char*.

Parameters **char** – Character to find the successor of.

Returns Successor character.

Example

```
>>> successor('b')
'c'
>>> successor('B')
'C'
>>> successor('2')
'3'
```

New in version 3.0.0.

pydash.strings.**surround**(*text: Any*, *wrapper: Any*) → str
Surround a string with another string.

Parameters

- **text** – String to surround with *wrapper*.
- **wrapper** – String by which *text* is to be surrounded.

Returns Surrounded string.

Example

```
>>> surround('abc', '"')
'"abc"'
>>> surround('abc', '!')
'!abc!'
```

New in version 2.4.0.

pydash.strings.**swap_case**(*text: Any*) → str
Swap case of *text* characters.

Parameters **text** – String to swap case.

Returns String with swapped case.

Example

```
>>> swap_case('aBcDeF')
'AbCdEf'
```

New in version 3.0.0.

pydash.strings.**title_case**(*text: Any*) → str
Convert *text* to title case.

Parameters **text** – String to convert.

Returns String converted to title case.

Example

```
>>> title_case("bob's shop")
"Bob's Shop"
```

New in version 3.0.0.

pydash.strings.**to_lower**(*text: Any*) → str
Converts the given `text` to lower text.

Parameters **text** – String to convert.

Returns String converted to lower case.

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: Any*) → str
Converts the given `text` to upper text.

Parameters **text** – String to convert.

Returns String converted to upper case.

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: Any*, *chars: Optional[str] = None*) → str
Removes leading and trailing whitespace or specified characters from *text*.

Parameters

- **text** – String to trim.
- **chars** – Specific characters to remove.

Returns Trimmed string.

Example

```
>>> trim('  abc efg\r\n ')
'abc efg'
```

New in version 1.1.0.

pydash.strings.**trim_end**(*text: Any*, *chars: Optional[str] = None*) → str
Removes trailing whitespace or specified characters from *text*.

Parameters

- **text** – String to trim.
- **chars** – Specific characters to remove.

Returns Trimmed string.

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: Any*, *chars: Optional[str] = None*) → str
Removes leading whitespace or specified characters from *text*.

Parameters

- **text** – String to trim.
- **chars** – Specific characters to remove.

Returns Trimmed string.

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: Any*, *length: int = 30*, *omission: str = '...'*, *separator: Optional[Union[str, re.Pattern]] = None*) → str
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** – String to truncate.
- **length** – Maximum string length. Defaults to `30`.
- **omission** – String to indicate text is omitted.
- **separator** – Separator pattern to truncate to.

Returns Truncated string.

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: Any*) → str

The inverse of *escape()*. This method converts the HTML entities `&`, `<`, `>`, `&quot;`, `&#39;`, and ``` in *text* to their corresponding characters.

Parameters **text** – String to unescape.

Returns HTML unescaped string.

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: Any*, *quote_char: Any = '"'*) → str

Unquote *text* by removing *quote_char* if *text* begins and ends with it.

Parameters

- **text** – String to unquote.
- **quote_char** – Quote character to remove. Defaults to ".

Returns Unquoted string.

Example

```
>>> unquote('"abc"')
'abc'
>>> unquote('"abc"', '#')
'"abc"'
>>> unquote('#abc', '#')
'#abc'
>>> unquote('#abc#', '#')
'abc'
```

New in version 3.0.0.

pydash.strings.**upper_case**(*text: Any*) → str
Converts string to upper case, as space separated words.

Parameters **text** – String to be converted to uppercase.

Returns String converted to uppercase, as space separated words.

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.

Changed in version 5.0.0: Improved unicode word support.

pydash.strings.**upper_first**(*text: str*) → str
Converts the first character of string to upper case.

Parameters **text** – String passed in by the user.

Returns String in which the first character is converted to upper case.

Example

```
>>> upper_first('fred')
'Fred'
>>> upper_first('foo bar')
'Foo bar'
>>> upper_first('1foobar')
'1foobar'
>>> upper_first(';foobar')
';foobar'
```

New in version 4.0.0.

pydash.strings.**url**(**paths: Any*, ***params: Any*) → str
Combines a series of URL paths into a single URL. Optionally, pass in keyword arguments to append query parameters.

Parameters **paths** – URL paths to combine.

Keyword Arguments **params** – Query parameters.

Returns URL string.

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: Any*, *pattern: Optional[str] = None*) → List[str]
Return list of words contained in *text*.

References

https://github.com/lodash/lodash/blob/master/words.js#L30

Parameters

- **text** – String to split.
- **pattern** – Custom pattern to split words on. Defaults to `None`.

Returns List of words.

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.

Changed in version 5.0.0: Improved unicode word support.

5.1.10 Utilities

Utility functions.

New in version 1.0.0.

pydash.utilities.**attempt**(*func: Callable[[pydash.utilities.P], pydash.utilities.T]*, **args: P.args*, ***kwargs: P.kwargs*) → Union[pydash.utilities.T, Exception]
Attempts to execute *func*, returning either the result or the caught error object.

Parameters **func** – The function to attempt.

Returns Returns the *func* result or error object.

Example

```
>>> results = attempt(lambda x: x/0, 1)
>>> assert isinstance(results, ZeroDivisionError)
```

New in version 1.1.0.

pydash.utilities.**cond**(*pairs: List[Tuple[Callable[[pydash.utilities.P], Any], Callable[[pydash.utilities.P], pydash.utilities.T]]]*, **extra_pairs: Tuple[Callable[[pydash.utilities.P], Any], Callable[[pydash.utilities.P], pydash.utilities.T]]*) → Callable[[pydash.utilities.P], pydash.utilities.T]

pydash.utilities.**cond**(*pairs: List[List[Callable[[pydash.utilities.P], Any]]]*, **extra_pairs: List[Callable[[pydash.utilities.P], Any]]*) → Callable[[pydash.utilities.P], Any]

Creates a function that iterates over *pairs* and invokes the corresponding function of the first predicate to return truthy.

Parameters **pairs** – A list of predicate-function pairs.

Returns Returns the new composite function.

Example

```
>>> func = cond([[matches({'a': 1}), constant('matches A')],
↪ [matches({'b': 2}), constant('matches B')],                    [stub_true,␣
↪lambda value: value]])
>>> func({'a': 1, 'b': 2})
'matches A'
>>> func({'a': 0, 'b': 2})
'matches B'
>>> func({'a': 0, 'b': 0}) == {'a': 0, 'b': 0}
True
```

New in version 4.0.0.

Changed in version 4.2.0: Fixed missing argument passing to matched function and added support for passing in a single list of pairs instead of just pairs as separate arguments.

pydash.utilities.**conforms**(*source: Dict[pydash.utilities.T, Callable[[pydash.utilities.T2], Any]]*) → Callable[[Dict[pydash.utilities.T, pydash.utilities.T2]], bool]

pydash.utilities.**conforms**(*source: List[Callable[[pydash.utilities.T], Any]]*) → Callable[[List[pydash.utilities.T]], bool]

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** – The object of property predicates to conform to.

Returns Returns the new spec 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: Dict[pydash.utilities.T, pydash.utilities.T2]*, *source: Dict[pydash.utilities.T, Callable[[pydash.utilities.T2], Any]]*) → bool
pydash.utilities.**conforms_to**(*obj: List[pydash.utilities.T]*, *source: List[Callable[[pydash.utilities.T], Any]]*) → bool

Checks if *obj* conforms to *source* by invoking the predicate properties of *source* with the corresponding property values of *obj*.

Parameters

- **obj** – The object to inspect.
- **source** – 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: pydash.utilities.T*) → Callable[[...], pydash.utilities.T]

Creates a function that returns *value*.

Parameters **value** – Constant value to return.

Returns Function that always returns *value*.

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: Optional[pydash.utilities.T]*, *default_value: pydash.utilities.T2*) → Union[pydash.utilities.T, pydash.utilities.T2]

Checks *value* to determine whether a default value should be returned in its place. The *default_value* is returned if value is None.

Parameters **default_value** – Default value passed in by the user.

Returns Returns *value* if `value` is given otherwise returns *default_value*.

Example

```
>>> default_to(1, 10)
1
>>> default_to(None, 10)
10
```

New in version 4.0.0.

pydash.utilities.**default_to_any**(*value: None*, **default_values: None*) → None
pydash.utilities.**default_to_any**(*value: Optional[pydash.utilities.T]*, *default_value1: None*, *default_value2: pydash.utilities.T2*) → Union[pydash.utilities.T, pydash.utilities.T2]
pydash.utilities.**default_to_any**(*value: Optional[pydash.utilities.T]*, *default_value1: None*, *default_value2: None*, *default_value3: pydash.utilities.T2*) → Union[pydash.utilities.T, pydash.utilities.T2]
pydash.utilities.**default_to_any**(*value: Optional[pydash.utilities.T]*, *default_value1: None*, *default_value2: None*, *default_value3: None*, *default_value4: pydash.utilities.T2*) → Union[pydash.utilities.T, pydash.utilities.T2]
pydash.utilities.**default_to_any**(*value: Optional[pydash.utilities.T]*, *default_value1: None*, *default_value2: None*, *default_value3: None*, *default_value4: None*, *default_value5: pydash.utilities.T2*) → Union[pydash.utilities.T, pydash.utilities.T2]
pydash.utilities.**default_to_any**(*value: Optional[pydash.utilities.T]*, **default_values: pydash.utilities.T2*) → Union[pydash.utilities.T, pydash.utilities.T2]

Checks *value* to determine whether a default value should be returned in its place. The first item that is not None of the *default_values* is returned.

Parameters

- **value** – Value passed in by the user.
- ***default_values** – Default values passed in by the user.

Returns

Returns *value* if `value` is given otherwise returns the first not None value of *default_values*.

Example

```
>>> default_to_any(1, 10, 20)
1
>>> default_to_any(None, 10, 20)
10
>>> default_to_any(None, None, 20)
20
```

New in version 4.9.0.

pydash.utilities.**identity**(*arg: pydash.utilities.T*, **args: Any*) → pydash.utilities.T
pydash.utilities.**identity**(*arg: None = None*, **args: Any*) → None

Return the first argument provided to it.

Parameters ***args** – Arguments.

Returns First argument or `None`.

Example

```
>>> identity(1)
1
>>> identity(1, 2, 3)
1
>>> identity() is None
True
```

New in version 1.0.0.

pydash.utilities.**iteratee**(*func: Callable[[pydash.utilities.P], pydash.utilities.T]*) →
Callable[[pydash.utilities.P], pydash.utilities.T]
pydash.utilities.**iteratee**(*func: Any*) → Callable

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** – Object to create iteratee function from.

Returns Iteratee 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})
```

(continues on next page)

(continued from previous page)

```
5
>>> iteratee('a.b')({'a': {'b': 5}})
5
>>> iteratee(('a', ['c', 'd', 'e']))({'a': 1, 'c': {'d': {'e': 3}}})
[1, 3]
>>> 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`.

Changed in version 4.1.0: Return *properties()* callback when *func* is a `tuple`.

pydash.utilities.**matches**(*source: Any*) → Callable[[Any], bool]

Creates a matches-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** – Source object used for comparision.

Returns

Function that compares an object to *source* and returns whether the two objects contain the same items.

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: Any*, *value: Any*) → Callable[[Any], bool]

Creates a function that compares the property value of *key* on a given object to *value*.

Parameters

- **key** – Object key to match against.
- **value** – Value to compare to.

Returns

Function that compares *value* to an object's *key* and returns whether they are equal.

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: Callable[[pydash.utilities.P], pydash.utilities.T], resolver: None = None*) → pydash.utilities.MemoizedFunc[pydash.utilities.P, pydash.utilities.T, str]

pydash.utilities.**memoize**(*func: Callable[[pydash.utilities.P], pydash.utilities.T], resolver: Optional[Callable[[pydash.utilities.P], pydash.utilities.T2]] = None*) → pydash.utilities.MemoizedFunc[pydash.utilities.P, pydash.utilities.T, pydash.utilities.T2]

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 to memoize.
- **resolver** – Function that returns the cache key to use.

Returns Memoized 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: Union[Hashable, List[Hashable]], *args: Any, **kwargs: Any*) → Callable[[...], Any]

Creates a function that invokes the method at *path* on a given object. Any additional arguments are provided to the invoked method.

Parameters

- **path** – Object path of method to invoke.

- ***args** – Global arguments to apply to method when invoked.
- ****kwargs** – Global keyword argument to apply to method when invoked.

Returns Function that invokes method located at path for object.

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: Any*, **args: Any*, ***kwargs: Any*) → Callable[[...], Any]
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** – The object to query.
- ***args** – Global arguments to apply to method when invoked.
- ****kwargs** – Global keyword argument to apply to method when invoked.

Returns Function that invokes method located at path for object.

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: Any*, ***kwargs: Any*) → None
A no-operation function.

New in version 1.0.0.

pydash.utilities.**now**() → int
Return the number of milliseconds that have elapsed since the Unix epoch (1 January 1970 00:00:00 UTC).

Returns Milliseconds since Unix epoch.

New in version 1.0.0.

Changed in version 3.0.0: Use `datetime` module for calculating elapsed time.

pydash.utilities.**nth_arg**(*pos: int = 0*) → Callable[[...], Any]
Creates a function that gets the argument at index n. If n is negative, the nth argument from the end is returned.

Parameters **pos** – The index of the argument to return.

Returns Returns the new pass-thru 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.**over**(*funcs: Iterable[Callable[[pydash.utilities.P], pydash.utilities.T]]*) → Callable[[pydash.utilities.P], List[pydash.utilities.T]]

Creates a function that invokes all functions in *funcs* with the arguments it receives and returns their results.

Parameters **funcs** – List of functions to be invoked.

Returns Returns the new pass-thru function.

Example

```
>>> func = over([max, min])
>>> func(1, 2, 3, 4)
[4, 1]
```

New in version 4.0.0.

pydash.utilities.**over_every**(*funcs: Iterable[Callable[[pydash.utilities.P], Any]]*) → Callable[[pydash.utilities.P], bool]

Creates a function that checks if all the functions in *funcs* return truthy when invoked with the arguments it receives.

Parameters **funcs** – List of functions to be invoked.

Returns Returns the new pass-thru 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: Iterable[Callable[[pydash.utilities.P], Any]]*) → Callable[[pydash.utilities.P], bool]

Creates a function that checks if any of the functions in *funcs* return truthy when invoked with the arguments it receives.

Parameters **funcs** – List of functions to be invoked.

Returns Returns the new pass-thru function.

Example

```
>>> func = over_some([bool, lambda x: x is None])
>>> func(1)
True
```

New in version 4.0.0.

pydash.utilities.**properties**(**paths: Any*) → Callable[[Any], Any]
Like *property_()* except that it returns a list of values at each path in *paths*.

Parameters ***path** – Path values to fetch from object.

Returns Function that returns object's path value.

Example

```
>>> getter = properties('a', 'b', ['c', 'd', 'e'])
>>> getter({'a': 1, 'b': 2, 'c': {'d': {'e': 3}}})
[1, 2, 3]
```

New in version 4.1.0.

pydash.utilities.**property_**(*path: Union[Hashable, List[Hashable]]*) → Callable[[Any], Any]
Creates a function that returns the value at path of a given object.

Parameters **path** – Path value to fetch from object.

Returns Function that returns object's path value.

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: Any*) → Callable[[Union[Hashable, List[Hashable]]], Any]
The inverse of *property_()*. This method creates a function that returns the key value of a given key on *obj*.

Parameters **obj** – Object to fetch values from.

Returns Function that returns object's key value.

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: int = 0*, *stop: int = 1*, ***, *floating: typing_extensions.Literal[False] = 'False'*) → int
pydash.utilities.**random**(*start: float*, *stop: int = 1*, *floating: bool = False*) → float
pydash.utilities.**random**(*start: int = 0*, ***, *stop: float*, *floating: bool = 'False'*) → float
pydash.utilities.**random**(*start: float*, *stop: float*, *floating: bool = False*) → float
pydash.utilities.**random**(*start: Union[float, int] = 0*, *stop: Union[float, int] = 1*, ***, *floating: typing_extensions.Literal[True]*) → float

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** – Minimum value.
- **stop** – Maximum value.
- **floating** – Whether to force random value to `float`. Defaults to `False`.

Returns Random value.

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_**(*stop: int*) → Generator[int, None, None]
pydash.utilities.**range_**(*start: int*, *stop: int*, *step: int = 1*) → Generator[int, None, None]

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** – Integer to start with. Defaults to `0`.
- **stop** – Integer to stop at.
- **step** – The value to increment or decrement by. Defaults to 1.

Yields 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.uilities`.

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**(*stop: int*) → Generator[int, None, None]
pydash.utilities.**range_right**(*start: int*, *stop: int*, *step: int = 1*) → Generator[int, None, None]

Similar to *range_()*, except that it populates the values in descending order.

Parameters

- **start** – Integer to start with. Defaults to `0`.
- **stop** – Integer to stop at.
- **step** – The value to increment or decrement by. Defaults to `1` if *start* < *stop* else `-1`.

Yields 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: None*, *key: Any*, *default: None = None*) → None
pydash.utilities.**result**(*obj: None*, *key: Any*, *default: pydash.utilities.T*) → pydash.utilities.T
pydash.utilities.**result**(*obj: Any*, *key: Any*, *default: Any = None*) → Any

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** – Object to retrieve result from.
- **key** – Key or index to get result from.
- **default** – Default value to return if *obj* is falsey. Defaults to `None`.

Returns Result of `obj[key]` or `None`.

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.**retry**(*attempts: int = 3, delay: Union[int, float] = 0.5, max_delay: Union[int, float] = 150.0, scale: Union[int, float] = 2.0, jitter: Union[int, float, Tuple[Union[float, int], Union[float, int]]] = 0, exceptions: Iterable[Type[Exception]] = (<class 'Exception'>,), on_exception: Optional[Callable[[Exception, int], Any]] = None*) → Callable[[pydash.utilities.CallableT], pydash.utilities.CallableT]

Decorator that retries a function multiple times if it raises an exception with an optional delay between each attempt.

When a *delay* is supplied, there will be a sleep period in between retry attempts. The first delay time will always be equal to *delay*. After subsequent retries, the delay time will be scaled by *scale* up to *max_delay*. If *max_delay* is `0`, then *delay* can increase unbounded.

Parameters

- **attempts** – Number of retry attempts. Defaults to `3`.
- **delay** – Base amount of seconds to sleep between retry attempts. Defaults to `0.5`.
- **max_delay** – Maximum number of seconds to sleep between retries. Is ignored when equal to `0`. Defaults to `150.0` (2.5 minutes).
- **scale** – Scale factor to increase *delay* after first retry fails. Defaults to `2.0`.
- **jitter** – Random jitter to add to *delay* time. Can be a positive number or 2-item tuple of numbers representing the random range to choose from. When a number is given, the random range will be from `[0, jitter]`. When jitter is a float or contains a float, then a random float will be chosen; otherwise, a random integer will be selected. Defaults to `0` which disables jitter.
- **exceptions** – Tuple of exceptions that trigger a retry attempt. Exceptions not in the tuple will be ignored. Defaults to `(Exception,)` (all exceptions).
- **on_exception** – Function that is called when a retryable exception is caught. It is invoked with `on_exception(exc, attempt)` where `exc` is the caught exception and `attempt` is the attempt count. All arguments are optional. Defaults to `None`.

Example

```
>>> @retry(attempts=3, delay=0)
... def do_something():
...     print('something')
...     raise Exception('something went wrong')
>>> try: do_something()
... except Exception: print('caught something')
something
something
something
caught something
```

..versionadded:: 4.4.0

..versionchanged:: 4.5.0 Added `jitter` argument.

pydash.utilities.**stub_dict**() → Dict
Returns empty "dict".

Returns Empty dict.

Example

```
>>> stub_dict()
{}
```

New in version 4.0.0.

pydash.utilities.**stub_false**() → typing_extensions.Literal[False]
Returns `False`.

Returns False

Example

```
>>> stub_false()
False
```

New in version 4.0.0.

pydash.utilities.**stub_list**() → List
Returns empty "list".

Returns Empty list.

Example

```
>>> stub_list()
[]
```

New in version 4.0.0.

pydash.utilities.**stub_string**() → str
Returns an empty string.

Returns Empty string

Example

```
>>> stub_string()
''
```

New in version 4.0.0.

pydash.utilities.**stub_true**() → typing_extensions.Literal[True]
Returns `True`.

Returns True

Example

```
>>> stub_true()
True
```

New in version 4.0.0.

pydash.utilities.**times**(*n: int*, *iteratee: Callable[[...], pydash.utilities.T]*) → List[pydash.utilities.T]
pydash.utilities.**times**(*n: int*, *iteratee: None = None*) → List[int]
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** – Number of times to execute *iteratee*.
- **iteratee** – Function to execute.

Returns A list of results from calling *iteratee*.

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: Union[Hashable, List[Hashable]]*) → List[Hashable]

Converts values to a property path array.

Parameters **value** – Value to convert.

Returns Returns the new property path array.

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.

Changed in version 4.2.1: Ensure returned path is always a list.

pydash.utilities.**unique_id**(*prefix: Optional[str] = None*) → str

Generates a unique ID. If *prefix* is provided the ID will be appended to it.

Parameters **prefix** – String prefix to prepend to ID value.

Returns ID value.

Example

```
>>> unique_id()
'1'
>>> unique_id('id_')
'id_2'
>>> unique_id()
'3'
```

New in version 1.0.0.

CHAPTER

SIX

PROJECT INFO

6.1 License

6.2 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`.

6.3 Changelog

6.3.1 v7.0.3 (2023-05-04)

- Fix typing for `difference_by`, `intersection_by`, `union_by`, `uniq_by`, and `xor_by` by allowing `iteratee` argument to be *Any*. Thanks DeviousStoat!

6.3.2 v7.0.2 (2023-04-27)

- Fix issue where using `pyright` as a type checker with `reportPrivateUsage=true` would report errors that objects are not exported from `pydash`. Thanks DeviousStoat!

6.3.3 v7.0.1 (2023-04-13)

- Fix missing install dependency, `typing-extensions`, for package.

6.3.4 v7.0.0 (2023-04-11)

- Add type annotations to package. Raise an issue for any typing issues at https://github.com/dgilland/pydash/issues. Thanks DeviousStoat! (**breaking change**)
- Change behavior of `to_dict` to not using `dict()` internally. Previous behavior would be for something like `to_dict([["k", "v"], ["x", "y"]])` to return `{"k": "v", "x": "y"}` (equivalent to calling `dict(...)`) but `to_dict([["k"], ["v"], ["x"], ["y"]])` would return `{0: ["x"], 1: ["v"], 2: ["x"], 3: ["y"]}`. The new behavior is to always return iterables as dictionaries with their indexes as keys like `{0: ["k", "v"], 1: ["x", "y"]}`. This is consistent with how iterable objects are iterated over and means that `to_dict` will have more reliable output. (**breaking change**)
- Change behavior of `slugify` to remove single-quotes from output. Instead of `slugify("the cat's meow") == "the-cat's-meow"`, the new behavior is to return `"the-cats-meow"`. (**breaking change**)
- Add support for negative indexes in `get` path keys. Thanks bl4ckst0ne!

6.3.5 v6.0.2 (2023-02-23)

- Only prevent access to object paths containing `__globals__` or `__builtins__` instead of all dunder-methods for non-dict/list objects.

6.3.6 v6.0.1 (2023-02-20)

- Fix exception raised due to mishandling of non-string keys in functions like `get()` for non-dict/list objects that used integer index references like `"[0]"`.

6.3.7 v6.0.0 (2023-01-28)

- Prevent access to object paths containing dunder-methods in functions like `get()` for non-dict/list objects. Attempting to access dunder-methods using get-path keys will raise a `KeyError` (e.g. `get(SomeClass(), '__init__'` will raise). Access to dict keys are unaffected (e.g. `get({"__init__": True}, "__init__")` will return `True`). (**breaking change**)
- Add support for Python 3.11.
- Drop support for Python 3.6 (**breaking change**)

6.3.8 v5.1.2 (2022-11-30)

- Remove unnecessary type check and conversion for `exceptions` argument in `pydash.retry`.

6.3.9 v5.1.1 (2022-09-23)

- Add support for Python 3.10.
- Fix timing assertion issue in test for `pydash.delay` where it could fail on certain environments.

6.3.10 v5.1.0 (2021-10-02)

- Support matches-style callbacks on non-dictionary objects that are compatible with `pydash.get` in functions like `pydash.find`.

6.3.11 v5.0.2 (2021-07-15)

- Fix compatibility issue between `pydash.py_` / `pydash._` and `typing.Protocol` + `typing.runtime_checkable` that caused an exception to be raised for `isinstance(py_, SomeRuntimeCheckableProtocol)`.

6.3.12 v5.0.1 (2021-06-27)

- Fix bug in `merge_with` that prevented custom iteratee from being used when recursively merging. Thanks weineel!

6.3.13 v5.0.0 (2021-03-29)

- Drop support for Python 2.7. (**breaking change**)
- Improve Unicode word splitting in string functions to be inline with Lodash. Thanks mervynlee94! (**breaking change**)
  - `camel_case`
  - `human_case`
  - `kebab_case`
  - `lower_case`
  - `pascal_case`

  - `separator_case`
  - `slugify`
  - `snake_case`
  - `start_case`
  - `upper_case`

- Optimize regular expression constants used in `pydash.strings` by pre-compiling them to regular expression pattern objects.

6.3.14 v4.9.3 (2021-03-03)

- Fix regression introduced in `v4.8.0` that caused `merge` and `merge_with` to raise an exception when passing `None` as the first argument.

6.3.15 v4.9.2 (2020-12-24)

- Fix regression introduced in `v4.9.1` that broke `pydash.get` for dictionaries and dot-delimited keys that reference integer dict-keys.

6.3.16 v4.9.1 (2020-12-14)

- Fix bug in `get/has` that caused `defaultdict` objects to get populated on key access.

6.3.17 v4.9.0 (2020-10-27)

- Add `default_to_any`. Thanks gonzalonaveira!
- Fix mishandling of key names containing `\.` in `set_`, `set_with`, and `update_with` where the `.` was not treated as a literal value within the key name. Thanks zhaowb!

6.3.18 v4.8.0 (2020-06-13)

- Support attribute based setters like `argparse.Namespace` in `set_`, `set_with`, `update`, and `update_with`.
- Fix exception in `order_by` when `None` used as a sort key. Thanks elijose55!
- Fix behavior of `pick_by` to return the passed in argument when only one argument given. Previously, an empty dictionary was returned. Thanks elijose55!
- Officially support Python 3.8.

6.3.19 v4.7.6 (2019-11-20)

Bug Fixes

- Fix handling of `Sequence`, `Mapping`, and `namedtuple` types in `get` so that their attributes aren't accessed during look-up. Thanks jwilson8767!

6.3.20 v4.7.5 (2019-05-21)

Bug Fixes

- Fix handling of string and byte values in `clone_with` and `clone_deep_with` when a customizer is used.
- Fix handling of non-indexable iterables in `find` and `find_last`.

6.3.21 v4.7.4 (2018-11-14)

Bug Fixes

- Raise an explicit exception in `set_`, `set_with`, `update`, and `update_with` when an object cannot be updated due to that object or one of its nested objects not being subscriptable.

6.3.22 v4.7.3 (2018-08-07)

Bug Fixes

- Fix bug in `spread` where arguments were not being passed to wrapped function properly.

6.3.23 v4.7.1 (2018-08-03)

New Features

- Modify `to_dict` to first try to convert using `dict()` before falling back to using `pydash.helpers.iterator()`.

6.3.24 v4.7.0 (2018-07-26)

Misc

- Internal code optimizations.

6.3.25 v4.6.1 (2018-07-16)

Misc

- Support Python 3.7.

6.3.26 v4.6.0 (2018-07-10)

Misc

- Improve performance of the following functions for large datasets:
  - `duplicates`
  - `sorted_uniq`
  - `sorted_uniq_by`
  - `union`
  - `union_by`
  - `union_with`
  - `uniq`
  - `uniq_by`
  - `uniq_with`
  - `xor`
  - `xor_by`
  - `xor_with`

6.3.27 v4.5.0 (2018-03-20)

New Features

- Add `jitter` argument to `retry`.

6.3.28 v4.4.1 (2018-03-14)

New Features

- Add `attempt` argument to `on_exception` callback in `retry`. New function signature is `on_exception(exc, attempt)` (previously was `on_exception(exc)`). All arguments to `on_exception` callback are now optional.

6.3.29 v4.4.0 (2018-03-13)

New Features

- Add `retry` decorator that will retry a function multiple times if the function raises an exception.

6.3.30 v4.3.3 (2018-03-02)

Bug Fixes

- Fix regression in `v4.3.2` introduced by the support added for callable class callbacks that changed the handling of callbacks that could not be inspected. Prior to `v4.3.2`, these callbacks would default to being passed a single callback argument, but with `v4.3.2` these callbacks would be passed the full set of callback arguments which could result an exception being raised due to the callback not supporting that many arguments.

6.3.31 v4.3.2 (2018-02-06)

Bug Fixes

- Fix issue in `defaults_deep` where sources with non-dict values would raise an exception due to assumption that object was always a dict.
- Fix issue in `curry` where too many arguments would be passed to the curried function when evaluating function if too many arguments used in last function call.
- Workaround issue in Python 2.7 where callable classes used as callbacks were always passed the full count of arguments even when the callable class only accept a subset of arguments.

6.3.32 v4.3.1 (2017-12-19)

Bug Fixes

- Fix `set_with` so that callable values are not called when being set. This bug also impacted the following functions by proxy:
  - `pick`
  - `pick_by`
  - `set_`
  - `transpose`
  - `zip_object_deep`

6.3.33 v4.3.0 (2017-11-22)

New Features

- Add `nest`.
- Wrap non-iterables in a list in `to_list` instead of raising an exception. Thanks efenka!
- Add `split_strings` argument to `to_list` to control whether strings are coverted to a list (`split_strings=True`) or wrapped in a list (`split_strings=False`). Default is `split_strings=True`. Thanks efenka!

6.3.34 v4.2.1 (2017-09-08)

Bug Fixes

- Ensure that `to_path` always returns a `list`.
- Fix `get` to work with path values other than just strings, integers, and lists.

6.3.35 v4.2.0 (2017-09-08)

New Features

- Support more iterator "hooks" in `to_dict` so non-iterators that expose an `items()`, `iteritems()`, or has `__dict__` attributes will be converted using those methods.
- Support deep paths in `omit` and `omit_by`. Thanks beck3905!
- Support deep paths in `pick` and `pick_by`. Thanks beck3905!

Bug Fixes

- Fix missing argument passing to matched function in `cond`.
- Support passing a single list of pairs in `cond` instead of just pairs as separate arguments.

6.3.36 v4.1.0 (2017-06-09)

New Features

- Officially support Python 3.6.
- Add `properties` function that returns list of path values for an object.
- Add `replace_end`.
- Add `replace_start`.
- Make `iteratee` support `properties`-style callback when a `tuple` is passed.
- Make `replace` accept `from_start` and `from_end` arguments to limit replacement to start and/or end of string.

Bug Fixes

- None

6.3.37 v4.0.4 (2017-05-31)

New Features

- None

Bug Fixes

- Improve performance of `get`. Thanks shaunpatterson!

6.3.38 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.

6.3.39 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

6.3.40 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_`).

6.3.41 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**)

6.3.42 v3.4.8 (2017-01-05)

- Make internal function inspection methods work with Python 3 annotations. Thanks tgriesser!

6.3.43 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 urbnjamesmi1!

6.3.44 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 urbnjamesmi1!

6.3.45 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`.

6.3.46 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`.

6.3.47 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.

6.3.48 v3.4.2 (2016-03-24)

- Fix bug in `start_case` where capitalized characters after the first character of a word where mistakenly cast to lower case.

6.3.49 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`.

6.3.50 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.

6.3.51 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.

6.3.52 v3.2.2 (2015-04-29)

- Catch `AttributeError` in `helpers.get_item` and return default value if set.

6.3.53 v3.2.1 (2015-04-29)

- Fix bug in `reduce_right` where collection was not reversed correctly.

6.3.54 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.

6.3.55 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,)`.

6.3.56 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.

6.3.57 v2.4.2 (2015-02-03)

- Fix `remove` so that array is modified after callback iteration.

6.3.58 v2.4.1 (2015-01-11)

- Fix `kebab_case` so that it casts string to lower case.

6.3.59 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!

6.3.60 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.

6.3.61 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 `_`.

6.3.62 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.

6.3.63 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`.

6.3.64 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`.

6.3.65 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`.

6.3.66 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**)

6.3.67 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`.

6.3.68 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`.

6.4 Authors

6.4.1 Lead

- Derrick Gilland, dgilland@gmail.com, dgilland@github

6.4.2 Contributors

- Nathan Cahill, nathan@nathancahill.com, nathancahill@github
- Klaus Sevensleeper, k7sleeper@gmail.com, k7sleeper@github
- Bharadwaj Yarlagadda, yarlagaddabharadwaj@gmail.com, bharadwajyarlagadda@github
- Michael James, urbnjamesmi1@github
- Tim Griesser, tgriesser@gmail.com, tgriesser@github
- Shaun Patterson, shaunpatterson@github
- Brian Beck, beck3905@github
- Frank Epperlein, efenka@github
- Joshua Wilson, jwilson8767@github
- Eli Jose, elijose55@github
- Gonzalo Naveira, gonzalonaveira@github
- Wenbo Zhao, zhaowb@gmail.com, zhaowb@github
- Mervyn Lee, mervynlee94@github
- Weineel Lee, weineel@github

6.5 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:

6.5.1 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 :)

6.5.2 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_username_here/pydash.git
```

3. Install Python dependencies into a virtualenv:

```
$ cd pydash
$ pip install -r requirements.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. Autoformat code:

```
$ inv fmt
```

6. When you're done making changes, check that your changes pass all unit tests by testing with `tox` across all supported Python versions:

```
$ tox
```

7. Add yourself to `AUTHORS.rst`.
8. 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-branch
```

9. Submit a pull request through GitHub.

6.5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. The pull request should work for all versions Python that this project supports.

6.6 Kudos

Thank you to Lodash for providing such a great library to port.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z