DeepDiff Documentation

Release 4.0.7

Sep Dehpour

Oct 12, 2021

CONTENTS

Note:

Visit Zepworks.com for the current documentations.

DeepDiff: Deep Difference of dictionaries, iterables, strings and other objects. It will recursively look for all the changes.

DeepSearch: Search for objects within other objects.

DeepHash: Hash any object based on their content even if they are not "hashable".

DeepDiff works with Python 3.4, 3.5, 3.6, 3.7, Pypy3

NOTE: Python 2 is not supported any more. DeepDiff v3.3.0 was the last version to supprt Python 2.

CHAPTER

ONE

INSTALLATION

Install from PyPi:

```
pip install deepdiff
```

DeepDiff prefers to use Murmur3 for hashing. However you have to manually install Murmur3 by running:

```
pip install 'deepdiff[murmur]'
```

Otherwise DeepDiff will be using SHA256 for hashing which is a cryptographic hash and is considerably slower.

If you are running into trouble installing Murmur3, please take a look at the Troubleshoot

1.1 Importing

```
>>> from deepdiff import DeepDiff  # For Deep Difference of 2 objects
>>> from deepdiff import grep, DeepSearch  # For finding if item exists in an object
>>> from deepdiff import DeepHash  # For hashing objects based on their contents
```

CHAPTER

TWO

DEEPDIFF

Note:

Visit Zepworks.com for the current documentations.

Read The DeepDiff details in:

DeepSearch

Short introduction

2.1 Supported data types

int, string, dictionary, list, tuple, set, frozenset, OrderedDict, NamedTuple and custom objects!

2.2 Ignore Order

Sometimes you don't care about the order of objects when comparing them. In those cases, you can set `ignore_order=True`. However this flag won't report the repetitions to you. You need to additionally enable `report_repetition=True` for getting a report of repetitions.

2.2.1 List difference ignoring order or duplicates

```
>>> t1 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 2, 3]}}
>>> t2 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 3, 2, 3]}}
>>> ddiff = DeepDiff(t1, t2, ignore_order=True)
>>> print (ddiff)
{}
```

2.3 Exclude types or paths

2.3.1 Exclude certain types from comparison

```
>>> l1 = logging.getLogger("test")
>>> l2 = logging.getLogger("test2")
>>> t1 = {"log": l1, 2: 1337}
>>> t2 = {"log": l2, 2: 1337}
>>> print(DeepDiff(t1, t2, exclude_types={logging.Logger}))
{}
```

2.4 Significant Digits

Digits **after** the decimal point. Internally it uses "{:.Xf}".format(Your Number) to compare numbers where X=significant_digits

```
>>> t1 = Decimal('1.52')
>>> t2 = Decimal('1.57')
>>> DeepDiff(t1, t2, significant_digits=0)
{}
>>> DeepDiff(t1, t2, significant_digits=1)
{'values_changed': {'root': {'old_value': Decimal('1.52'), 'new_value': Decimal('1.57')}}
↪}
```

2.5 Serialization

Serialize to json

```
>>> t1 = {1: 1, 2: 2, 3: 3}
>>> t2 = {1: 1, 2: "2", 3: 3}
>>> ddiff = DeepDiff(t1, t2)
>>> jsoned = ddiff.to_json()
>>> jsoned
'{"type_changes": {"root[2]": {"new_type": "str", "new_value": "2", "old_type": "int",
↪"old_value": 2}}}'
```

And many more features! Read more in

DeepSearch

CHAPTER

THREE

DEEP SEARCH

Note:

Visit Zepworks.com for the current documentations.

Deep Search inside objects to find the item matching your criteria.

Note that is searches for either the path to match your criteria or the word in an item.

Examples

Importing

```
>>> from deepdiff import DeepSearch, grep
>>> from pprint import pprint
```

DeepSearch comes with grep function which is easier to remember!

Search in list for string

```
>>> obj = ["long somewhere", "string", 0, "somewhere great!"]
>>> item = "somewhere"
>>> ds = obj | grep(item, verbose_level=2)
>>> print(ds)
{'matched_values': {'root[3]': 'somewhere great!', 'root[0]': 'long somewhere'}}
```

Search in nested data for string

```
>>> obj = ["something somewhere", {"long": "somewhere", "string": 2, 0: 0, "somewhere":
↪"around"}]
>>> item = "somewhere"
>>> ds = obj | grep(item, verbose_level=2)
>>> pprint(ds, indent=2)
{ 'matched_paths': {"root[1]['somewhere']": 'around'},
  'matched_values': { 'root[0]': 'something somewhere',
                      "root[1]['long']": 'somewhere'}}
```

Read more in the Deep Search references:

DeepSearch

CHAPTER

FOUR

DEEP HASH

Note:

Visit Zepworks.com for the current documentations.

DeepHash calculates the hash of objects based on their contents in a deterministic way. This way 2 objects with the same content should have the same hash.

The main usage of DeepHash is to calculate the hash of otherwise unhashable objects. For example you can use DeepHash to calculate the hash of a set or a dictionary!

The core of DeepHash is a deterministic serialization of your object into a string so it can be passed to a hash function. By default it uses Murmur 3 128 bit hash function. but you can pass another hash function to it if you want.

Read the details at:

DeepHash

Examples:

Let's say you have a dictionary object.

```
>>> from deepdiff import DeepHash
>>>
>>> obj = {1: 2, 'a': 'b'}
```

If you try to hash it:

```
>>> hash(obj)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

But with DeepHash:

```
>>> from deepdiff import DeepHash
>>> obj = {1: 2, 'a': 'b'}
>>> DeepHash(obj)
{1: 24689164770724817775122835877892927 49, 2: -3578777349255665377637755521812243 1491, ..
↪.}
```

So what is exactly the hash of obj in this case? DeepHash is calculating the hash of the obj and any other object that obj contains. The output of DeepHash is a dictionary of object IDs to their hashes. In order to get the hash of obj itself, you need to use the object (or the id of object) to get its hash:

```
>>> hashes = DeepHash(obj)
>>> hashes[obj]
341508986457500994779872229399128149852
```

Read more in the Deep Hash reference:

DeepHash

CHAPTER

FIVE

TROUBLESHOOT

5.1 Murmur3

Failed to build mmh3 when installing DeepDiff

DeepDiff prefers to use Murmur3 for hashing. However you have to manually install murmur3 by running: *pip install mmh3*

On MacOS Mojave some user experience difficulty when installing Murmur3.

The problem can be solved by running:

xcode-select –install

And then running

pip install mmh3

CHAPTER

SIX

REFERENCES

DeepDiff OLD 4.0.7 documentation!

6.1 DeepDiff Reference

Note:

Visit Zepworks.com for the current documentations.

class deepdiff.diff.**DeepDiff**(*t1*, *t2*, *ignore_order=False*, *report_repetition=False*, *significant_digits=None*, *number_format_notation='f'*, *exclude_paths=None*, *exclude_regex_paths=None*, *exclude_types=None*, *ignore_type_in_groups=None*, *ignore_string_type_changes=False*, *ignore_numeric_type_changes=False*, *ignore_type_subclasses=False*, *ignore_string_case=False*, *number_to_string_func=None*, *verbose_level=1*, *view='text'*, *hasher=None*, ***kwargs*)

DeepDiff

Note: DeepDiff documentations are now hosted on Zepworks.com

What you see here are the old documentations.

Deep Difference of dictionaries, iterables, strings and almost any other object. It will recursively look for all the changes.

DeepDiff 3.0 added the concept of views. There is a default "text" view and a "tree" view.

Parameters

t1 [A dictionary, list, string or any python object that has __dict__ or __slots__] This is the first item to be compared to the second item

t2 [dictionary, list, string or almost any python object that has __dict__ or __slots__] The second item is to be compared to the first one

ignore_order [Boolean, defalt=False] ignores orders for iterables Note that if you have iterables contatining any unhashable, ignoring order can be expensive. Normally ignore_order does not report duplicates and repetition changes. In order to report repetitions, set report_repetition=True in addition to ignore_order=True

report_repetition [Boolean, default=False] reports repetitions when set True ONLY when ignore_order is set True too. This works for iterables. This feature currently is experimental and is not production ready.

significant_digits [int >= 0, default=None] By default the significant_digits compares only that many digits AFTER the decimal point. However you can set override that by setting the number_format_notation="e" which will make it mean the digits in scientific notation.

Important: This will affect ANY number comparison when it is set.

Note: If ignore_numeric_type_changes is set to True and you have left significant_digits to the default of None, it gets automatically set to 55. The reason is that normally when numbers from 2 different types are compared, instead of comparing the values, we only report the type change. However when ignore_numeric_type_changes=True, in order compare numbers from different types to each other, we need to convert them all into strings. The significant_digits will be used to make sure we accurately convert all the numbers into strings in order to report the changes between them.

Internally it uses "{:.Xf}".format(Your Number) to compare numbers where X=significant_digits when the number_format_notation is left as the default of "f" meaning fixed point.

Note that "{:.3f}".format(1.1135) = 1.113, but "{:.3f}".format(1.11351) = 1.114

For Decimals, Python's format rounds 2.5 to 2 and 3.5 to 4 (to the closest even number)

When you set the number_format_notation="e", we use "{:.Xe}".format(Your Number) where X=significant_digits.

number_format_notation [string, default="f"] number_format_notation is what defines the meaning of significant digits. The default value of "f" means the digits AFTER the decimal point. "f" stands for fixed point. The other option is "e" which stands for exponent notation or scientific notation.

number_to_string_func [function, default=None] This is an advanced feature to give the user the full control into overriding how numbers are converted to strings for comparison. The default function is defined in https://github.com/seperman/deepdiff/blob/master/deepdiff/helper.py and is called number_to_string. You can define your own function to do that.

verbose_level: int >= 0, default = 1 Higher verbose level shows you more details. For example verbose level 1 shows what dictionary item are added or removed. And verbose level 2 shows the value of the items that are added or removed too.

exclude_paths: list, default = None List of paths to exclude from the report. If only one item, you can path it as a string.

exclude_regex_paths: list, default = None List of string regex paths or compiled regex paths objects to exclude from the report. If only one item, you can pass it as a string or regex compiled object.

hasher: default = DeepHash.murmur3_128bit Hash function to be used. If you don't want Murmur3, you can use Python's built-in hash function by passing hasher=hash. This is for advanced usage and normally you don't need to modify it.

view: string, default = text Starting the version 3 you can choosethe view into the deepdiff results. The default is the text view which has been the only view up until now. The new view is called the tree view which allows you to traverse through the tree of changed items.

exclude_types: list, default = None List of object types to exclude from the report.

ignore_string_type_changes: Boolean, default = False Whether to ignore string type changes or not. For example b"Hello" vs. "Hello" are considered the same if ignore_string_type_changes is set to True.

ignore_numeric_type_changes: Boolean, default = False Whether to ignore numeric type changes or not. For example 10 vs. 10.0 are considered the same if ignore_numeric_type_changes is set to True.

ignore_type_in_groups: Tuple or List of Tuples, default = None ignores types when t1 and t2 are both within the same type group.

ignore_type_subclasses: Boolean, default = False ignore type (class) changes when dealing with the subclasses of classes that were marked to be ignored.

ignore_string_case: Boolean, default = False Whether to be case-sensitive or not when comparing strings. By settings ignore_string_case=False, strings will be compared case-insensitively.

Returns

A DeepDiff object that has already calculated the difference of the 2 items.

Supported data types

int, string, unicode, dictionary, list, tuple, set, frozenset, OrderedDict, NamedTuple and custom objects!

Text View

Text view is the original and currently the default view of DeepDiff.

It is called text view because the results contain texts that represent the path to the data:

Example of using the text view.

```
>>> from deepdiff import DeepDiff
>>> t1 = {1:1, 3:3, 4:4}
>>> t2 = {1:1, 3:3, 5:5, 6:6}
>>> ddiff = DeepDiff(t1, t2)
>>> print(ddiff)
{'dictionary_item_added': [root[5], root[6]], 'dictionary_item_removed':␣
↪[root[4]]}
```

So for example ddiff['dictionary_item_added'] is a set of strings thus this is called the text view.

See also:

The following examples are using the *default text view.* The Tree View is introduced in DeepDiff v3 and provides traversing capabilitie through your diffed data and more! Read more about the Tree View at the bottom of this page.

Importing

```
>>> from deepdiff import DeepDiff
>>> from pprint import pprint
```

Same object returns empty

```
>>> t1 = {1:1, 2:2, 3:3}
>>> t2 = t1
>>> print(DeepDiff(t1, t2))
{}
```

Type of an item has changed

```
>>> t1 = {1:1, 2:2, 3:3}
>>> t2 = {1:1, 2:"2", 3:3}
>>> pprint(DeepDiff(t1, t2), indent=2)
{ 'type_changes': { 'root[2]': { 'new_type': <class 'str'>,
                                 'new_value': '2',
                                 'old_type': <class 'int'>,
                                 'old_value': 2}}}
```

Value of an item has changed

```
>>> t1 = {1:1, 2:2, 3:3}
>>> t2 = {1:1, 2:4, 3:3}
>>> pprint(DeepDiff(t1, t2, verbose_level=0), indent=2)
{'values_changed': {'root[2]': {'new_value': 4, 'old_value': 2}}}
```

Item added and/or removed

```
>>> t1 = {1:1, 3:3, 4:4}
>>> t2 = {1:1, 3:3, 5:5, 6:6}
>>> ddiff = DeepDiff(t1, t2)
>>> pprint (ddiff)
{'dictionary_item_added': [root[5], root[6]],
 'dictionary_item_removed': [root[4]]}
```

Set verbose level to 2 in order to see the added or removed items with their values

```
>>> t1 = {1:1, 3:3, 4:4}
>>> t2 = {1:1, 3:3, 5:5, 6:6}
>>> ddiff = DeepDiff(t1, t2, verbose_level=2)
>>> pprint(ddiff, indent=2)
{ 'dictionary_item_added': {'root[5]': 5, 'root[6]': 6},
  'dictionary_item_removed': {'root[4]': 4}}
```

String difference

```
>>> t1 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":"world"}}
>>> t2 = {1:1, 2:4, 3:3, 4:{"a":"hello", "b":"world!"}}
>>> ddiff = DeepDiff(t1, t2)
>>> pprint (ddiff, indent = 2)
{ 'values_changed': { 'root[2]': {'new_value': 4, 'old_value': 2},
                      "root[4]['b']": { 'new_value': 'world!',
                                        'old_value': 'world'}}}
```

String difference 2

```
>>> t1 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":"world!\nGoodbye!\n1\n2\nEnd"}}
>>> t2 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":"world\n1\n2\nEnd"}}
>>> ddiff = DeepDiff(t1, t2)
>>> pprint (ddiff, indent = 2)
{ 'values_changed': { "root[4]['b']": { 'diff': '--- \n'
                                              '+++ \n'
                                              '@@ -1,5 +1,4 @@\n'
                                              '-world!\n'
                                              '-Goodbye!\n'
                                              '+world\n'
                                              ' 1\n'
                                              ' 2\n'
                                              ' End',
                                        'new_value': 'world\n1\n2\nEnd',
                                        'old_value': 'world!\n'
                                                     'Goodbye!\n'
                                                     '1\n'
                                                     '2\n'
                                                     'End'}}}
```

```
>>>
>>> print (ddiff['values_changed']["root[4]['b']"]["diff"])
---
+++
@@ -1,5 +1,4 @@
-world!
-Goodbye!
+world
 1
 2
 End
```

List difference

```
>>> t1 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 2, 3, 4]}}
>>> t2 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 2]}}
>>> ddiff = DeepDiff(t1, t2)
>>> pprint (ddiff, indent = 2)
{'iterable_item_removed': {"root[4]['b'][2]": 3, "root[4]['b'][3]": 4}}
```

List difference 2:

```
>>> t1 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 2, 3]}}
>>> t2 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 3, 2, 3]}}
>>> ddiff = DeepDiff(t1, t2)
>>> pprint (ddiff, indent = 2)
{ 'iterable_item_added': {"root[4]['b'][3]": 3},
  'values_changed': { "root[4]['b'][1]": {'new_value': 3, 'old_value': 2},
                      "root[4]['b'][2]": {'new_value': 2, 'old_value': 3}}}
```

List difference ignoring order or duplicates: (with the same dictionaries as above)

```
>>> t1 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 2, 3]}}
>>> t2 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 3, 2, 3]}}
>>> ddiff = DeepDiff(t1, t2, ignore_order=True)
>>> print (ddiff)
{}
```

List difference ignoring order but reporting repetitions:

```
>>> from deepdiff import DeepDiff
>>> from pprint import pprint
>>> t1 = [1, 3, 1, 4]
>>> t2 = [4, 4, 1]
>>> ddiff = DeepDiff(t1, t2, ignore_order=True, report_repetition=True)
>>> pprint(ddiff, indent=2)
{ 'iterable_item_removed': {'root[1]': 3},
  'repetition_change': { 'root[0]': { 'new_indexes': [2],
                                      'new_repeat': 1,
                                      'old_indexes': [0, 2],
                                      'old_repeat': 2,
                                      'value': 1},
                         'root[3]': { 'new_indexes': [0, 1],
```

(continues on next page)

(continued from previous page)

```
                                        'new_repeat': 2,
                                        'old_indexes': [3],
                                        'old_repeat': 1,
                                        'value': 4}}}
```

List that contains dictionary:

```
>>> t1 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 2, {1:1, 2:2}]}}
>>> t2 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 2, {1:3}]}}
>>> ddiff = DeepDiff(t1, t2)
>>> pprint (ddiff, indent = 2)
{ 'dictionary_item_removed': [root[4]['b'][2][2]],
  'values_changed': {"root[4]['b'][2][1]": {'new_value': 3, 'old_value': 1}}}
```

Sets:

```
>>> t1 = {1, 2, 8}
>>> t2 = {1, 2, 3, 5}
>>> ddiff = DeepDiff(t1, t2)
>>> pprint(ddiff)
{'set_item_added': [root[3], root[5]], 'set_item_removed': [root[8]]}
```

Named Tuples:

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> t1 = Point(x=11, y=22)
>>> t2 = Point(x=11, y=23)
>>> pprint (DeepDiff(t1, t2))
{'values_changed': {'root.y': {'new_value': 23, 'old_value': 22}}}
```

Custom objects:

```
>>> class ClassA(object):
...     a = 1
...     def __init__(self, b):
...         self.b = b
...
>>> t1 = ClassA(1)
>>> t2 = ClassA(2)
>>>
>>> pprint(DeepDiff(t1, t2))
{'values_changed': {'root.b': {'new_value': 2, 'old_value': 1}}}
```

Object attribute added:

```
>>> t2.c = "new attribute"
>>> pprint(DeepDiff(t1, t2))
{'attribute_added': [root.c],
 'values_changed': {'root.b': {'new_value': 2, 'old_value': 1}}}
```

Approximate decimals comparison (Significant digits after the point):

```
>>> t1 = Decimal('1.52')
>>> t2 = Decimal('1.57')
>>> DeepDiff(t1, t2, significant_digits=0)
{}
>>> DeepDiff(t1, t2, significant_digits=1)
{'values_changed': {'root': {'new_value': Decimal('1.57'), 'old_value': Decimal(
↪'1.52')}}}
```

Approximate float comparison (Significant digits after the point):

```
>>> t1 = [ 1.1129, 1.3359 ]
>>> t2 = [ 1.113, 1.3362 ]
>>> pprint(DeepDiff(t1, t2, significant_digits=3))
{}
>>> pprint(DeepDiff(t1, t2))
{'values_changed': {'root[0]': {'new_value': 1.113, 'old_value': 1.1129},
                    'root[1]': {'new_value': 1.3362, 'old_value': 1.3359}}}
>>> pprint(DeepDiff(1.23*10**20, 1.24*10**20, significant_digits=1))
{'values_changed': {'root': {'new_value': 1.24e+20, 'old_value': 1.23e+20}}}
```

Approximate number comparison (significant_digits after the decimal point in scientific notation)

```
>>> DeepDiff(1024, 1020, significant_digits=2, number_format_notation="f")  #␣
↪default is "f"
{'values_changed': {'root': {'new_value': 1020, 'old_value': 1024}}}
>>> DeepDiff(1024, 1020, significant_digits=2, number_format_notation="e")
{}
```

Defining your own number_to_string_func Lets say you want the numbers comparison happen only for numbers above 100 for some reason.

```
>>> from deepdiff import DeepDiff
>>> from deepdiff.helper import number_to_string
>>> def custom_number_to_string(number, *args, **kwargs):
...     number = 100 if number < 100 else number
...     return number_to_string(number, *args, **kwargs)
...
>>> t1 = [10, 12, 100000]
>>> t2 = [50, 63, 100021]
>>> DeepDiff(t1, t2, significant_digits=3, number_format_notation="e")
{'values_changed': {'root[0]': {'new_value': 50, 'old_value': 10}, 'root[1]': {
↪'new_value': 63, 'old_value': 12}}}
>>>
>>> DeepDiff(t1, t2, significant_digits=3, number_format_notation="e",
...          number_to_string_func=custom_number_to_string)
{}
```

Note: All the examples for the text view work for the tree view too. You just need to set view='tree' to get it in tree form.

Ignore Type Changes

Type change

```
>>> t1 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 2, 3]}}
>>> t2 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":"world\n\n\nEnd"}}
>>> ddiff = DeepDiff(t1, t2)
>>> pprint (ddiff, indent = 2)
{ 'type_changes': { "root[4]['b']": { 'new_type': <class 'str'>,
                                      'new_value': 'world\n\n\nEnd',
                                      'old_type': <class 'list'>,
                                      'old_value': [1, 2, 3]}}}
```

And if you don't care about the value of items that have changed type, please set verbose level to 0

```
>>> t1 = {1:1, 2:2, 3:3}
>>> t2 = {1:1, 2:"2", 3:3}
>>> pprint(DeepDiff(t1, t2, verbose_level=0), indent=2)
{ 'type_changes': { 'root[2]': { 'new_type': <class 'str'>,
                                 'old_type': <class 'int'>}}}
```

Exclude types

Exclude certain types from comparison:

```
>>> l1 = logging.getLogger("test")
>>> l2 = logging.getLogger("test2")
>>> t1 = {"log": l1, 2: 1337}
>>> t2 = {"log": l2, 2: 1337}
>>> print(DeepDiff(t1, t2, exclude_types={logging.Logger}))
{}
```

ignore_type_in_groups Ignore type changes between members of groups of types. For example if you want to ignore type changes between float and decimals etc. Note that this is a more granular feature. Most of the times the shortcuts provided to you are enough. The shortcuts are ignore_string_type_changes which by default is False and ignore_numeric_type_changes which is by default False. You can read more about those shortcuts in this page. ignore_type_in_groups gives you more control compared to the shortcuts.

For example lets say you have specifically str and byte datatypes to be ignored for type changes. Then you have a couple of options:

1. Set ignore_string_type_changes=True.
2. Or set ignore_type_in_groups=[(str, bytes)]. Here you are saying if we detect one type to be str and the other one bytes, do not report them as type change. It is exactly as passing ignore_type_in_groups=[DeepDiff.strings] or ignore_type_in_groups=DeepDiff.strings .

Now what if you want also typeA and typeB to be ignored when comparing agains each other?

1. ignore_type_in_groups=[DeepDiff.strings, (typeA, typeB)]
2. or ignore_type_in_groups=[(str, bytes), (typeA, typeB)]

ignore_string_type_changes Default: False

```
>>> DeepDiff(b'hello', 'hello', ignore_string_type_changes=True)
{}
>>> DeepDiff(b'hello', 'hello')
{'type_changes': {'root': {'old_type': <class 'bytes'>, 'new_type': <class 'str
↪'>, 'old_value': b'hello', 'new_value': 'hello'}}}
```

ignore_numeric_type_changes Default: False Ignore Type Number - Dictionary that contains float and integer >>> from deepdiff import DeepDiff >>> from pprint import pprint >>> t1 = {1: 1, 2: 2.22} >>> t2 = {1: 1.0, 2: 2.22} >>> ddiff = DeepDiff(t1, t2) >>> pprint(ddiff, indent=2) { 'type_changes': { 'root[1]': { 'new_type': <class 'float'>,

'new_value': 1.0, 'old_type': <class 'int'>, 'old_value': 1}}}

```
>>> ddiff = DeepDiff(t1, t2, ignore_type_in_groups=DeepDiff.numbers)
>>> pprint(ddiff, indent=2)
{}
```

Ignore Type Number - List that contains float and integer

```
>>> from deepdiff import DeepDiff
>>> from pprint import pprint
>>> t1 = [1, 2, 3]
>>> t2 = [1.0, 2.0, 3.0]
>>> ddiff = DeepDiff(t1, t2)
>>> pprint(ddiff, indent=2)
{ 'type_changes': { 'root[0]': { 'new_type': <class 'float'>,
                                 'new_value': 1.0,
                                 'old_type': <class 'int'>,
                                 'old_value': 1},
                    'root[1]': { 'new_type': <class 'float'>,
                                 'new_value': 2.0,
                                 'old_type': <class 'int'>,
                                 'old_value': 2},
                    'root[2]': { 'new_type': <class 'float'>,
                                 'new_value': 3.0,
                                 'old_type': <class 'int'>,
                                 'old_value': 3}}}
>>> ddiff = DeepDiff(t1, t2, ignore_type_in_groups=DeepDiff.numbers)
>>> pprint(ddiff, indent=2)
{}
```

You can pass a list of tuples or list of lists if you have various type groups. When t1 and t2 both fall under one of these type groups, the type change will be ignored. DeepDiff already comes with 2 groups: DeepDiff.strings and DeepDiff.numbers . If you want to pass both: >>> ignore_type_in_groups = [DeepDiff.strings, DeepDiff.numbers]

ignore_type_in_groups example with custom objects:

```
>>> class Burrito:
...     bread = 'flour'
...     def __init__(self):
...         self.spicy = True
...
>>>
>>> class Taco:
...     bread = 'flour'
...     def __init__(self):
...         self.spicy = True
...
>>>
>>> burrito = Burrito()
```

(continues on next page)

(continued from previous page)

```
>>> taco = Taco()
>>>
>>> burritos = [burrito]
>>> tacos = [taco]
>>>
>>> DeepDiff(burritos, tacos, ignore_type_in_groups=[(Taco, Burrito)], ignore_
↪order=True)
{}
```

ignore_type_subclasses Use ignore_type_subclasses=True so when ignoring type (class), the subclasses of that class are ignored too.

```
>>> from deepdiff import DeepDiff
>>> class ClassA:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> class ClassB:
...     def __init__(self, x):
...         self.x = x
...
>>> class ClassC(ClassB):
...     pass
...
>>> obj_a = ClassA(1, 2)
>>> obj_c = ClassC(3)
>>>
>>> DeepDiff(obj_a, obj_c, ignore_type_in_groups=[(ClassA, ClassB)], ignore_
↪type_subclasses=False)
{'type_changes': {'root': {'old_type': <class '__main__.ClassA'>, 'new_type':
↪<class '__main__.ClassC'>, 'old_value': <__main__.ClassA object at␣
↪0x10076a2e8>, 'new_value': <__main__.ClassC object at 0x10082f630>}}}
>>>
>>> DeepDiff(obj_a, obj_c, ignore_type_in_groups=[(ClassA, ClassB)], ignore_
↪type_subclasses=True)
{'values_changed': {'root.x': {'new_value': 3, 'old_value': 1}}, 'attribute_
↪removed': [root.y]}
```

ignore_string_case Whether to be case-sensitive or not when comparing strings. By settings ignore_string_case=False, strings will be compared case-insensitively.

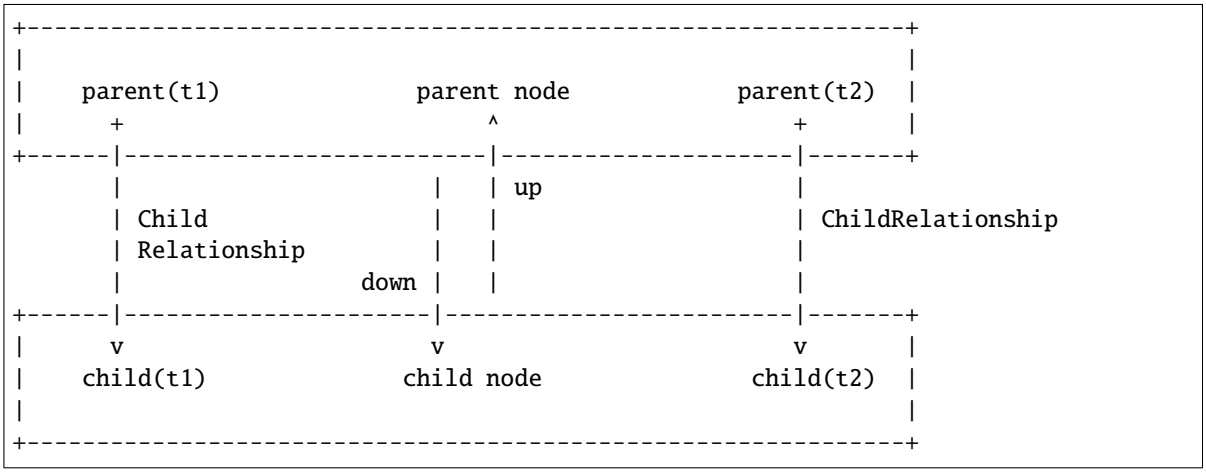
```
>>> DeepDiff(t1='Hello', t2='heLLO')
{'values_changed': {'root': {'new_value': 'heLLO', 'old_value': 'Hello'}}}
>>> DeepDiff(t1='Hello', t2='heLLO', ignore_string_case=True)
{}
```

Tree View

Starting the version 3 You can chooe the view into the deepdiff results. The tree view provides you with tree objects that you can traverse through to find the parents of the objects that are diffed and the actual objects that are being diffed. This view is very useful when dealing with nested objects. Note that tree view always returns results in the form of Python sets.

You can traverse through the tree elements!

Note: The Tree view is just a different representation of the diffed data. Behind the scene, DeepDiff creates the tree view first and then converts it to textual representation for the text view.



Up Move up to the parent node

Down Move down to the child node

Path() Get the path to the current node

T1 The first item in the current node that is being diffed

T2 The second item in the current node that is being diffed

Additional Additional information about the node i.e. repetition

Repetition Shortcut to get the repetition report

The tree view allows you to have more than mere textual representaion of the diffed objects. It gives you the actual objects (t1, t2) throughout the tree of parents and children.

Examples Tree View

Note: The Tree View is introduced in DeepDiff 3. Set view='tree' in order to use this view.

Value of an item has changed (Tree View)

```
>>> from deepdiff import DeepDiff
>>> from pprint import pprint
>>> t1 = {1:1, 2:2, 3:3}
>>> t2 = {1:1, 2:4, 3:3}
>>> ddiff_verbose0 = DeepDiff(t1, t2, verbose_level=0, view='tree')
>>> ddiff_verbose0
{'values_changed': [<root[2]>]}
>>>
>>> ddiff_verbose1 = DeepDiff(t1, t2, verbose_level=1, view='tree')
>>> ddiff_verbose1
```

(continues on next page)

(continued from previous page)

```
{'values_changed': [<root[2] t1:2, t2:4>]}
>>> set_of_values_changed = ddiff_verbose1['values_changed']
>>> # since set_of_values_changed includes only one item in a set
>>> # in order to get that one item we can:
>>> (changed,) = set_of_values_changed
>>> changed  # Another way to get this is to do: changed=list(set_of_values_
↪changed)[0]
<root[2] t1:2, t2:4>
>>> changed.t1
2
>>> changed.t2
4
>>> # You can traverse through the tree, get to the parents!
>>> changed.up
<root t1:{1: 1, 2: 2,...}, t2:{1: 1, 2: 4,...}>
```

List difference (Tree View)

```
>>> t1 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 2, 3, 4]}}
>>> t2 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 2]}}
>>> ddiff = DeepDiff(t1, t2, view='tree')
>>> ddiff
{'iterable_item_removed': [<root[4]['b'][2] t1:3, t2:not present>, <root[4]['b
↪'][3] t1:4, t2:not present>]}
>>> # Note that the iterable_item_removed is a set. In this case it has 2 items␣
↪in it.
>>> # One way to get one item from the set is to convert it to a list
>>> # And then get the first item of the list:
>>> removed = list(ddiff['iterable_item_removed'])[0]
>>> removed
<root[4]['b'][2] t1:3, t2:not present>
>>>
>>> parent = removed.up
>>> parent
<root[4]['b'] t1:[1, 2, 3, 4], t2:[1, 2]>
>>> parent.path()
"root[4]['b']"
>>> parent.t1
[1, 2, 3, 4]
>>> parent.t2
[1, 2]
>>> parent.up
<root[4] t1:{'a': 'hello...}, t2:{'a': 'hello...}>
>>> parent.up.up
<root t1:{1: 1, 2: 2,...}, t2:{1: 1, 2: 2,...}>
>>> parent.up.up.t1
{1: 1, 2: 2, 3: 3, 4: {'a': 'hello', 'b': [1, 2, 3, 4]}}
>>> parent.up.up.t1 == t1  # It is holding the original t1 that we passed to␣
↪DeepDiff
True
```

List difference 2 (Tree View)

```
>>> t1 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 2, 3]}}
>>> t2 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 3, 2, 3]}}
>>> ddiff = DeepDiff(t1, t2, view='tree')
>>> pprint(ddiff, indent = 2)
{ 'iterable_item_added': [<root[4]['b'][3] t1:not present, t2:3>],
  'values_changed': [<root[4]['b'][1] t1:2, t2:3>, <root[4]['b'][2] t1:3, t2:2>
↪]}
>>>
>>> # Note that iterable_item_added is a set with one item.
>>> # So in order to get that one item from it, we can do:
>>>
>>> (added,) = ddiff['iterable_item_added']
>>> added
<root[4]['b'][3] t1:not present, t2:3>
>>> added.up.up
<root[4] t1:{'a': 'hello...}, t2:{'a': 'hello...}>
>>> added.up.up.path()
'root[4]'
>>> added.up.up.down
<root[4]['b'] t1:[1, 2, 3], t2:[1, 3, 2, 3]>
>>>
>>> # going up twice and then down twice gives you the same node in the tree:
>>> added.up.up.down.down == added
True
```

List difference ignoring order but reporting repetitions (Tree View)

```
>>> t1 = [1, 3, 1, 4]
>>> t2 = [4, 4, 1]
>>> ddiff = DeepDiff(t1, t2, ignore_order=True, report_repetition=True, view=
↪'tree')
>>> pprint(ddiff, indent=2)
{ 'iterable_item_removed': [<root[1] t1:3, t2:not present>],
  'repetition_change': [<root[3] {'repetition': {'old_repeat': 1,...}>,
↪<root[0] {'repetition': {'old_repeat': 2,...}>]}
>>>
>>> # repetition_change is a set with 2 items.
>>> # in order to get those 2 items, we can do the following.
>>> # or we can convert the set to list and get the list items.
>>> # or we can iterate through the set items
>>>
>>> (repeat1, repeat2) = ddiff['repetition_change']
>>> repeat1  # the default verbosity is set to 1.
<root[3] {'repetition': {'old_repeat': 1,...}>
>>> # The actual data regarding the repetitions can be found in the repetition␣
↪attribute:
>>> repeat1.repetition
{'old_repeat': 1, 'new_repeat': 2, 'old_indexes': [3], 'new_indexes': [0, 1]}
>>>
>>> # If you change the verbosity, you will see less:
>>> ddiff = DeepDiff(t1, t2, ignore_order=True, report_repetition=True, view=
↪'tree', verbose_level=0)
>>> ddiff
```

(continues on next page)

(continued from previous page)

```
{'repetition_change': [<root[3]>, <root[0]>], 'iterable_item_removed': [
↪<root[1]>]}
>>> (repeat1, repeat2) = ddiff['repetition_change']
>>> repeat1
<root[0]>
>>>
>>> # But the verbosity level does not change the actual report object.
>>> # It only changes the textual representaion of the object. We get the␣
↪actual object here:
>>> repeat1.repetition
{'old_repeat': 1, 'new_repeat': 2, 'old_indexes': [3], 'new_indexes': [0, 1]}
>>> repeat1.t1
4
>>> repeat1.t2
4
>>> repeat1.up
<root>
```

List that contains dictionary (Tree View)

```
>>> t1 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 2, {1:1, 2:2}]}}
>>> t2 = {1:1, 2:2, 3:3, 4:{"a":"hello", "b":[1, 2, {1:3}]}}
>>> ddiff = DeepDiff(t1, t2, view='tree')
>>> pprint (ddiff, indent = 2)
{ 'dictionary_item_removed': [<root[4]['b'][2][2] t1:2, t2:not present>],
  'values_changed': [<root[4]['b'][2][1] t1:1, t2:3>]}
```

Sets (Tree View):

```
>>> t1 = {1, 2, 8}
>>> t2 = {1, 2, 3, 5}
>>> ddiff = DeepDiff(t1, t2, view='tree')
>>> print(ddiff)
{'set_item_removed': [<root: t1:8, t2:not present>], 'set_item_added': [<root:␣
↪t1:not present, t2:3>, <root: t1:not present, t2:5>]}
>>> # grabbing one item from set_item_removed set which has one item only
>>> (item,) = ddiff['set_item_removed']
>>> item.up
<root t1:{8, 1, 2}, t2:{1, 2, 3, 5}>
>>> item.up.t1 == t1
True
```

Named Tuples (Tree View):

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> t1 = Point(x=11, y=22)
>>> t2 = Point(x=11, y=23)
>>> print(DeepDiff(t1, t2, view='tree'))
{'values_changed': [<root.y t1:22, t2:23>]}
```

Custom objects (Tree View):

```
>>> class ClassA(object):
...     a = 1
...     def __init__(self, b):
...         self.b = b
...
>>> t1 = ClassA(1)
>>> t2 = ClassA(2)
>>>
>>> print(DeepDiff(t1, t2, view='tree'))
{'values_changed': [<root.b t1:1, t2:2>]}
```

Object attribute added (Tree View):

```
>>> t2.c = "new attribute"
>>> pprint(DeepDiff(t1, t2, view='tree'))
{'attribute_added': [<root.c t1:not present, t2:'new attribute'>],
 'values_changed': [<root.b t1:1, t2:2>]}
```

Approximate decimals comparison (Significant digits after the point) (Tree View):

```
>>> t1 = Decimal('1.52')
>>> t2 = Decimal('1.57')
>>> DeepDiff(t1, t2, significant_digits=0, view='tree')
{}
>>> ddiff = DeepDiff(t1, t2, significant_digits=1, view='tree')
>>> ddiff
{'values_changed': [<root t1:Decimal('1.52'), t2:Decimal('1.57')>]}
>>> (change1,) = ddiff['values_changed']
>>> change1
<root t1:Decimal('1.52'), t2:Decimal('1.57')>
>>> change1.t1
Decimal('1.52')
>>> change1.t2
Decimal('1.57')
>>> change1.path()
'root'
```

Approximate float comparison (Significant digits after the point) (Tree View):

```
>>> t1 = [ 1.1129, 1.3359 ]
>>> t2 = [ 1.113, 1.3362 ]
>>> ddiff = DeepDiff(t1, t2, significant_digits=3, view='tree')
>>> ddiff
{}
>>> ddiff = DeepDiff(t1, t2, view='tree')
>>> pprint(ddiff, indent=2)
{ 'values_changed': [<root[0] t1:1.1129, t2:1.113>, <root[1] t1:1.3359, t2:1.
↪3362>]}
>>> ddiff = DeepDiff(1.23*10**20, 1.24*10**20, significant_digits=1, view='tree
↪')
>>> ddiff
{'values_changed': [<root t1:1.23e+20, t2:1.24e+20>]}
```

Exclude paths

Exclude part of your object tree from comparison use *exclude_paths* and pass a set or list of paths to exclude, if only one item is being passed, then just put it there as a string. No need to pass it as a list then.

```
>>> t1 = {"for life": "vegan", "ingredients": ["no meat", "no eggs", "no dairy"]}
>>> t2 = {"for life": "vegan", "ingredients": ["veggies", "tofu", "soy sauce"]}
>>> print (DeepDiff(t1, t2, exclude_paths="root['ingredients']"))  # one item pass␣
↪it as a string
{}
>>> print (DeepDiff(t1, t2, exclude_paths=["root['ingredients']", "root[
↪'ingredients2']"]))  # multiple items pass as a list or a set.
{}
```

You can also exclude using regular expressions by using *exclude_regex_paths* and pass a set or list of path regexes to exclud

```
>>> import re
>>> t1 = [{'a': 1, 'b': 2}, {'c': 4, 'b': 5}]
>>> t2 = [{'a': 1, 'b': 3}, {'c': 4, 'b': 5}]
>>> print(DeepDiff(t1, t2, exclude_regex_paths=r"root\[\d+\]\['b'\]"))
{}
>>> exclude_path = re.compile(r"root\[\d+\]\['b'\]")
>>> print(DeepDiff(t1, t2, exclude_regex_paths=[exclude_path]))
{}
```

example 2:

```
>>> t1 = {'a': [1, 2, [3, {'foo1': 'bar'}]]}
>>> t2 = {'a': [1, 2, [3, {'foo2': 'bar'}]]}
>>> DeepDiff(t1, t2, exclude_regex_paths="\['foo.'\]")  # since it is one item␣
↪in exclude_regex_paths, you don't have to put it in a list or a set.
{}
```

Tip: DeepDiff is using re.search on the path. So if you want to force it to match from the beginning of the path, add ^ to the beginning of regex.

Note: All the examples for the text view work for the tree view too. You just need to set view='tree' to get it in tree form.

Serialization

In order to convert the DeepDiff object into a normal Python dictionary, use the to_dict() method. Note that to_dict will use the text view even if you did the diff in tree view.

Example:

```
>>> t1 = {1: 1, 2: 2, 3: 3, 4: {"a": "hello", "b": [1, 2, 3]}}
>>> t2 = {1: 1, 2: 2, 3: 3, 4: {"a": "hello", "b": "world\n\n\nEnd"}}
>>> ddiff = DeepDiff(t1, t2, view='tree')
>>> ddiff.to_dict()
{'type_changes': {"root[4]['b']": {'old_type': <class 'list'>, 'new_type':
↪<class 'str'>, 'old_value': [1, 2, 3], 'new_value': 'world\n\n\nEnd'}}}
```

In order to do safe json serialization, use the to_json() method.

Example:

```
>>> t1 = {1: 1, 2: 2, 3: 3, 4: {"a": "hello", "b": [1, 2, 3]}}
>>> t2 = {1: 1, 2: 2, 3: 3, 4: {"a": "hello", "b": "world\n\n\nEnd"}}
>>> ddiff = DeepDiff(t1, t2, view='tree')
>>> ddiff.to_json()
'{"type_changes": {"root[4][\'b\']": {"old_type": "list", "new_type": "str",
↪"old_value": [1, 2, 3], "new_value": "world\\n\\n\\nEnd"}}}'
```

See also:

Take a look at to_json() documentation in this page for more details.

If you want the original DeepDiff object to be serialized with all the bells and whistles, you can use the to_json_pickle() and to_json_pickle() in order to serialize and deserialize its results into json. Note that json_pickle is unsafe and json pickle dumps from untrusted sources should never be loaded.

Serialize and then deserialize back to deepdiff

```
>>> t1 = {1: 1, 2: 2, 3: 3}
>>> t2 = {1: 1, 2: "2", 3: 3}
>>> ddiff = DeepDiff(t1, t2)
>>> jsoned = ddiff.to_json_pickle()
>>> jsoned
'{"type_changes": {"root[2]": {"new_type": {"py/type": "builtins.str"}, "new_
↪value": "2", "old_type": {"py/type": "builtins.int"}, "old_value": 2}}}'
>>> ddiff_new = DeepDiff.from_json_pickle(jsoned)
>>> ddiff == ddiff_new
True
```

Pycon 2016 Talk I gave a talk about how DeepDiff does what it does at Pycon 2016. Diff it to Dig it Pycon 2016 video

And here is more info: http://zepworks.com/blog/diff-it-to-digg-it/

classmethod from_json_pickle(*value*)
Load DeepDiff object with all the bells and whistles from the json pickle dump. Note that json pickle dump comes from to_json_pickle

to_dict()
Dump dictionary of the text view. It does not matter which view you are currently in. It will give you the dictionary of the text view.

to_json(*default_mapping=None*)
Dump json of the text view. **Parameters**

default_mapping : default_mapping, dictionary(optional), a dictionary of mapping of different types to json types.

by default DeepDiff converts certain data types. For example Decimals into floats so they can be exported into json. If you have a certain object type that the json serializer can not serialize it, please pass the appropriate type conversion through this dictionary.

Example

Serialize custom objects

```
>>> class A:
...     pass
```

(continues on next page)

(continued from previous page)

```
...
>>> class B:
...     pass
...
>>> t1 = A()
>>> t2 = B()
>>> ddiff = DeepDiff(t1, t2)
>>> ddiff.to_json()
TypeError: We do not know how to convert <__main__.A object at 0x10648> of␣
↪type <class '__main__.A'> for json serialization. Please pass the default_
↪mapping parameter with proper mapping of the object to a basic python␣
↪type.
```

```
>>> default_mapping = {A: lambda x: 'obj A', B: lambda x: 'obj B'}
>>> ddiff.to_json(default_mapping=default_mapping)
'{"type_changes": {"root": {"old_type": "A", "new_type": "B", "old_value":
↪"obj A", "new_value": "obj B"}}}'
```

to_json_pickle()
Get the json pickle of the diff object. Unless you need all the attributes and functionality of DeepDiff, running to_json() is the safer option that json pickle.

Back to *DeepDiff OLD 4.0.7 documentation!*

DeepDiff OLD 4.0.7 documentation!

6.2 DeepSearch Reference

Note:

Visit Zepworks.com for the current documentations.

---

class deepdiff.search.**grep**(*item*, ***kwargs*)

Grep

Note: DeepDiff documentations are now hosted on Zepworks.com

What you see here are the old documentations.

---

grep is a new interface for Deep Search. It takes exactly the same arguments. And it works just like grep in shell!

Examples

Importing

```
>>> from deepdiff import grep
>>> from pprint import pprint
```

Search in list for string

```
>>> obj = ["long somewhere", "string", 0, "somewhere great!"]
>>> item = "somewhere"
>>> ds = obj | grep(item)
>>> print(ds)
{'matched_values': {'root[3]', 'root[0]'}
```

Search in nested data for string

```
>>> obj = ["something somewhere", {"long": "somewhere", "string": 2, 0: 0,
↪"somewhere": "around"}]
>>> item = "somewhere"
>>> ds = obj | grep(item, verbose_level=2)
>>> pprint(ds, indent=2)
{ 'matched_paths': {"root[1]['somewhere']": 'around'},
  'matched_values': { 'root[0]': 'something somewhere',
                      "root[1]['long']": 'somewhere'}}
```

class deepdiff.search.**DeepSearch**(*obj*, *item*, *exclude_paths={}*, *exclude_regex_paths={}*, *exclude_types={}*, *verbose_level=1*, *case_sensitive=False*, *match_string=False*, ***kwargs*)

DeepSearch

Deep Search inside objects to find the item matching your criteria.

Parameters

obj : The object to search within

item : The item to search for

verbose_level [int >= 0, default = 1.] Verbose level one shows the paths of found items. Verbose level 2 shows the path and value of the found items.

exclude_paths: list, default = None. List of paths to exclude from the report.

exclude_types: list, default = None. List of object types to exclude from the report.

case_sensitive: Boolean, default = False

match_string: Boolean, default = False If True, the value of the object or its children have to exactly match the item. If False, the value of the item can be a part of the value of the object or its children

Returns

A DeepSearch object that has the matched paths and matched values.

Supported data types

int, string, unicode, dictionary, list, tuple, set, frozenset, OrderedDict, NamedTuple and custom objects!

Examples

Importing

```
>>> from deepdiff import DeepSearch
>>> from pprint import pprint
```

Search in list for string

```
>>> obj = ["long somewhere", "string", 0, "somewhere great!"]
>>> item = "somewhere"
>>> ds = DeepSearch(obj, item, verbose_level=2)
```

(continues on next page)

(continued from previous page)

```
>>> print(ds)
{'matched_values': {'root[3]': 'somewhere great!', 'root[0]': 'long somewhere'}}
```

Search in nested data for string

```
>>> obj = ["something somewhere", {"long": "somewhere", "string": 2, 0: 0,
↪"somewhere": "around"}]
>>> item = "somewhere"
>>> ds = DeepSearch(obj, item, verbose_level=2)
>>> pprint(ds, indent=2)
{ 'matched_paths': {"root[1]['somewhere']": 'around'},
  'matched_values': { 'root[0]': 'something somewhere',
                      "root[1]['long']": 'somewhere'}}
```

Back to *DeepDiff OLD 4.0.7 documentation!*

DeepDiff OLD 4.0.7 documentation!

6.3 DeepHash Reference

Note:

Visit Zepworks.com for the current documentations.

class deepdiff.deephash.**DeepHash**(*obj*, ***, *hashes=None*, *exclude_types=None*, *exclude_paths=None*, *exclude_regex_paths=None*, *hasher=None*, *ignore_repetition=True*, *significant_digits=None*, *number_format_notation='f'*, *apply_hash=True*, *ignore_type_in_groups=None*, *ignore_string_type_changes=False*, *ignore_numeric_type_changes=False*, *ignore_type_subclasses=False*, *ignore_string_case=False*, *number_to_string_func=None*, ***kwargs*)

DeepHash

Note: DeepDiff documentations are now hosted on Zepworks.com

What you see here are the old documentations.

DeepHash calculates the hash of objects based on their contents in a deterministic way. This way 2 objects with the same content should have the same hash.

The main usage of DeepHash is to calculate the hash of otherwise unhashable objects. For example you can use DeepHash to calculate the hash of a set or a dictionary!

At the core of it, DeepHash is a deterministic serialization of your object into a string so it can be passed to a hash function. By default it uses Murmur 3 128 bit hash function which is a fast, non-cryptographic hashing function. You have the option to pass any another hashing function to be used instead.

If it can't find Murmur3 package (mmh3) installed, it uses Python's built-in SHA256 for hashing which is considerably slower than Murmur3. So it is advised that you install Murmur3 by running *pip install 'deepdiff[murmur]*

Import

```
>>> from deepdiff import DeepHash
```

Parameters

obj : any object, The object to be hashed based on its content.

hashes: dictionary, default = empty dictionary A dictionary of {object or object id: object hash} to start with. Any object that is encountered and it is already in the hashes dictionary or its id is in the hashes dictionary, will re-use the hash that is provided by this dictionary instead of re-calculating its hash. This is typically used when you have a series of objects to be hashed and there might be repeats of the same object.

exclude_types: list, default = None List of object types to exclude from hashing.

exclude_paths: list, default = None List of paths to exclude from the report. If only one item, you can path it as a string instead of a list containing only one path.

exclude_regex_paths: list, default = None List of string regex paths or compiled regex paths objects to exclude from the report. If only one item, you can path it as a string instead of a list containing only one regex path.

hasher: function. default = DeepHash.murmur3_128bit hasher is the hashing function. The default is DeepHash.murmur3_128bit. But you can pass another hash function to it if you want. For example a cryptographic hash function or Python's builtin hash function. All it needs is a function that takes the input in string format and returns the hash.

You can use it by passing: hasher=hash for Python's builtin hash.

The following alternatives are already provided:

- hasher=DeepHash.murmur3_128bit
- hasher=DeepHash.murmur3_64bit
- hasher=DeepHash.sha1hex

ignore_repetition: Boolean, default = True If repetitions in an iterable should cause the hash of iterable to be different. Note that the deepdiff diffing functionality lets this to be the default at all times. But if you are using DeepHash directly, you can set this parameter.

significant_digits [int >= 0, default=None] By default the significant_digits compares only that many digits AFTER the decimal point. However you can set override that by setting the number_format_notation="e" which will make it mean the digits in scientific notation.

Important: This will affect ANY number comparison when it is set.

Note: If ignore_numeric_type_changes is set to True and you have left significant_digits to the default of None, it gets automatically set to 12. The reason is that normally when numbers from 2 different types are compared, instead of comparing the values, we only report the type change. However when ignore_numeric_type_changes=True, in order compare numbers from different types to each other, we need to convert them all into strings. The significant_digits will be used to make sure we accurately convert all the numbers into strings in order to report the changes between them.

Internally it uses "{:.Xf}".format(Your Number) to compare numbers where X=significant_digits when the number_format_notation is left as the default of "f" meaning fixed point.

Note that "{:.3f}".format(1.1135) = 1.113, but "{:.3f}".format(1.11351) = 1.114

For Decimals, Python's format rounds 2.5 to 2 and 3.5 to 4 (to the closest even number)

When you set the number_format_notation="e", we use "{:.Xe}".format(Your Number) where X=significant_digits.

number_format_notation [string, default="f"] number_format_notation is what defines the meaning of significant digits. The default value of "f" means the digits AFTER the decimal point. "f" stands for fixed point. The other option is "e" which stands for exponent notation or scientific notation.

apply_hash: Boolean, default = True DeepHash at its core is doing deterministic serialization of objects into strings. Then it hashes the string. The only time you want the apply_hash to be False is if you want to know what the string representation of your object is BEFORE it gets hashed.

ignore_type_in_groups Ignore type changes between members of groups of types. For example if you want to ignore type changes between float and decimals etc. Note that this is a more granular feature. Most of the times the shortcuts provided to you are enough. The shortcuts are ignore_string_type_changes which by default is False and ignore_numeric_type_changes which is by default False. You can read more about those shortcuts in this page. ignore_type_in_groups gives you more control compared to the shortcuts.

For example lets say you have specifically str and byte datatypes to be ignored for type changes. Then you have a couple of options:

1. Set ignore_string_type_changes=True which is the default.
2. Set ignore_type_in_groups=[(str, bytes)]. Here you are saying if we detect one type to be str and the other one bytes, do not report them as type change. It is exactly as passing ignore_type_in_groups=[DeepDiff.strings] or ignore_type_in_groups=DeepDiff.strings .

Now what if you want also typeA and typeB to be ignored when comparing agains each other?

1. ignore_type_in_groups=[DeepDiff.strings, (typeA, typeB)]
2. or ignore_type_in_groups=[(str, bytes), (typeA, typeB)]

ignore_string_type_changes: Boolean, default = True string type conversions should not affect the hash output when this is set to True. For example "Hello" and b"Hello" should produce the same hash.

By setting it to True, both the string and bytes of hello return the same hash.

ignore_numeric_type_changes: Boolean, default = False numeric type conversions should not affect the hash output when this is set to True. For example 10, 10.0 and Decimal(10) should produce the same hash. When ignore_numeric_type_changes is set to True, all numbers are converted to strings with the precision of significant_digits parameter and number_format_notation notation. If no significant_digits is passed by the user, a default value of 12 is used.

ignore_type_subclasses Use ignore_type_subclasses=True so when ignoring type (class), the subclasses of that class are ignored too.

ignore_string_case Whether to be case-sensitive or not when comparing strings. By settings ignore_string_case=False, strings will be compared case-insensitively.

Returns A dictionary of {item: item hash}. If your object is nested, it will build hashes of all the objects it contains too.

Examples

Let's say you have a dictionary object.

```
>>> from deepdiff import DeepHash
>>> obj = {1: 2, 'a': 'b'}
```

If you try to hash it:

```
>>> hash(obj)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

But with DeepHash:

```
>>> from deepdiff import DeepHash
>>> obj = {1: 2, 'a': 'b'}
>>> DeepHash(obj)
{1: 234041559348429806012597903916437026784, 2:␣
↪148655924348182454950690728321917595655, 'a':␣
↪119173504597196970070553896747624927922, 'b':␣
↪499482722743792999173807660719621025 2, '!>*id4488569408':␣
↪324528384164125006864220932742479687 54}
```

So what is exactly the hash of obj in this case? DeepHash is calculating the hash of the obj and any other object that obj contains. The output of DeepHash is a dictionary of object IDs to their hashes. In order to get the hash of obj itself, you need to use the object (or the id of object) to get its hash:

```
>>> hashes = DeepHash(obj)
>>> hashes[obj]
341508986457500994779872229399128149852
```

Which you can write as:

```
>>> hashes = DeepHash(obj)[obj]
```

At first it might seem weird why DeepHash(obj)[obj] but remember that DeepHash(obj) is a dictionary of hashes of all other objects that obj contains too.

The result hash is 341508986457500994779872229399128149852 which is generated by Murmur 3 128bit hashing algorithm. If you prefer to use another hashing algorithm, you can pass it using the hasher parameter. Read more about Murmur3 here: https://en.wikipedia.org/wiki/MurmurHash

If you do a deep copy of obj, it should still give you the same hash:

```
>>> from copy import deepcopy
>>> obj2 = deepcopy(obj)
>>> DeepHash(obj2)[obj2]
341508986457500994779872229399128149852
```

Note that by default DeepHash will include string type differences. So if your strings were bytes:

```
>>> obj3 = {1: 2, b'a': b'b'}
>>> DeepHash(obj3)[obj3]
640675257658460244881039331016212127 60
```

But if you want the same hash if string types are different, set ignore_string_type_changes to True:

```
>>> DeepHash(obj3, ignore_string_type_changes=True)[obj3]
341508986457500994779872229399128149852
```

ignore_numeric_type_changes is by default False too.

```
>>> obj1 = {4:10}
>>> obj2 = {4.0: Decimal(10.0)}
>>> DeepHash(obj1)[4] == DeepHash(obj2)[4.0]
False
```

But by setting it to True, we can get the same hash.

```
>>> DeepHash(obj1, ignore_numeric_type_changes=True)[4] == DeepHash(obj2,␣
↪ignore_numeric_type_changes=True)[4.0]
True
```

number_format_notation: String, default = "f" number_format_notation is what defines the meaning of significant digits. The default value of "f" means the digits AFTER the decimal point. "f" stands for fixed point. The other option is "e" which stands for exponent notation or scientific notation.

ignore_string_type_changes: Boolean, default = True By setting it to True, both the string and bytes of hello return the same hash.

```
>>> DeepHash(b'hello', ignore_string_type_changes=True)
{b'hello': 221860156526691709602818861774599422448}
>>> DeepHash('hello', ignore_string_type_changes=True)
{'hello': 221860156526691709602818861774599422448}
```

ignore_numeric_type_changes: Boolean, default = False For example if significant_digits=5, 1.1, Decimal(1.1) are both converted to 1.10000

That way they both produce the same hash.

```
>>> t1 = {1: 1, 2: 2.22}
>>> t2 = {1: 1.0, 2: 2.22}
>>> DeepHash(t1)[1]
231678797214551245419120414857003063149
>>> DeepHash(t1)[1.0]
231678797214551245419120414857003063149
```

You can pass a list of tuples or list of lists if you have various type groups. When t1 and t2 both fall under one of these type groups, the type change will be ignored. DeepDiff already comes with 2 groups: DeepDiff.strings and DeepDiff.numbers . If you want to pass both:

```
>>> from deepdiff import DeepDiff
>>> ignore_type_in_groups = [DeepDiff.strings, DeepDiff.numbers]
```

ignore_type_in_groups example with custom objects:

```
>>> class Burrito:
...     bread = 'flour'
...     def __init__(self):
...         self.spicy = True
...
>>>
>>> class Taco:
...     bread = 'flour'
...     def __init__(self):
...         self.spicy = True
...
>>>
>>> burrito = Burrito()
>>> taco = Taco()
>>>
>>> burritos = [burrito]
```

(continues on next page)

(continued from previous page)

```
>>> tacos = [taco]
>>>
>>> d1 = DeepHash(burritos, ignore_type_in_groups=[(Taco, Burrito)])
>>> d2 = DeepHash(tacos, ignore_type_in_groups=[(Taco, Burrito)])
>>> d1[burrito] == d2[taco]
True
```

ignore_type_subclasses Use ignore_type_subclasses=True so when ignoring type (class), the subclasses of that class are ignored too.

```
>>> from deepdiff import DeepHash
>>>
>>> class ClassB:
...     def __init__(self, x):
...         self.x = x
...     def __repr__(self):
...         return "obj b"
...
>>>
>>> class ClassC(ClassB):
...     def __repr__(self):
...         return "obj c"
...
>>> obj_b = ClassB(1)
>>> obj_c = ClassC(1)
>>>
>>> # Since these 2 objects are from 2 different classes, the hashes are␣
↪different by default.
... # ignore_type_in_groups is set to [(ClassB, )] which means to ignore any␣
↪type conversion between
... # objects of classB and itself which does not make sense but it illustrates␣
↪a better point when
... # ignore_type_subclasses is set to be True.
... hashes_b = DeepHash(obj_b, ignore_type_in_groups=[(ClassB, )])
>>> hashes_c = DeepHash(obj_c, ignore_type_in_groups=[(ClassB, )])
>>> hashes_b[obj_b] != hashes_c[obj_c]
True
>>>
>>> # Hashes of these 2 objects will be the same when ignore_type_subclasses is␣
↪set to True
... hashes_b = DeepHash(obj_b, ignore_type_in_groups=[(ClassB, )], ignore_type_
↪subclasses=True)
>>> hashes_c = DeepHash(obj_c, ignore_type_in_groups=[(ClassB, )], ignore_type_
↪subclasses=True)
>>> hashes_b[obj_b] == hashes_c[obj_c]
True
```

ignore_string_case Whether to be case-sensitive or not when comparing strings. By settings ignore_string_case=False, strings will be compared case-insensitively.

```
>>> from deepdiff import DeepHash
>>> DeepHash('hello')['hello'] == DeepHash('heLLO')['heLLO']
```

(continues on next page)

(continued from previous page)

```
False
>>> DeepHash('hello', ignore_string_case=True)['hello'] == DeepHash('heLLO',␣
↪ignore_string_case=True)['heLLO']
True
```

number_format_notation [string, default="f"] When numbers are converted to the string, you have the choices between "f" as fixed point and "e" as scientific notation:

```
>>> t1=10002
>>> t2=10004
>>> t1_hash = DeepHash(t1, significant_digits=3, number_format_notation="f")
>>> t2_hash = DeepHash(t2, significant_digits=3, number_format_notation="f")
>>>
>>> t1_hash[t1] == t2_hash[t2]
False
>>>
>>>
>>> # Now we use the scientific notation
... t1_hash = DeepHash(t1, significant_digits=3, number_format_notation="e")
>>> t2_hash = DeepHash(t2, significant_digits=3, number_format_notation="e")
>>>
>>> t1_hash[t1] == t2_hash[t2]
True
```

Defining your own number_to_string_func Lets say you want the hash of numbers below 100 to be the same for some reason.

```
>>> from deepdiff import DeepHash
>>> from deepdiff.helper import number_to_string
>>> def custom_number_to_string(number, *args, **kwargs):
...     number = 100 if number < 100 else number
...     return number_to_string(number, *args, **kwargs)
...
>>> t1 = [10, 12, 100000]
>>> t2 = [50, 63, 100021]
>>> t1_hash = DeepHash(t1, significant_digits=3, number_format_notation="e",␣
↪number_to_string_func=custom_number_to_string)
>>> t2_hash = DeepHash(t2, significant_digits=3, number_format_notation="e",␣
↪number_to_string_func=custom_number_to_string)
>>> t1_hash[t1] == t2_hash[t2]
True
```

So both lists produced the same hash thanks to the low significant digits for 100000 vs 100021 and also the custom_number_to_string that converted all numbers below 100 to be 100!

static murmur3_128bit(*obj*)
Use murmur3_128bit for bit hash by passing this method: hasher=DeepHash.murmur3_128bit This hasher is the default hasher.

static murmur3_64bit(*obj*)
Use murmur3_64bit for 64 bit hash by passing this method: hasher=DeepHash.murmur3_64bit

static sha1hex(*obj*)
Use Sha1 as a cryptographic hash.

static sha256hex(*obj*)
Use Sha256 as a cryptographic hash.

Back to *DeepDiff OLD 4.0.7 documentation!*

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

EIGHT

CHANGELOG

- v4-0-7: Hashing of the number 1 vs. True
- v4-0-6: found a tiny bug in Python formatting of numbers in scientific notation. Added a workaround.
- v4-0-5: Fixing number diffing. Adding number_format_notation and number_to_string_func.
- v4-0-4: Adding ignore_string_case and ignore_type_subclasses
- v4-0-3: Adding versionbump tool for release
- v4-0-2: Fixing installation issue where rst files are missing.
- v4-0-1: Fixing installation Tarball missing requirements.txt . DeepDiff v4+ should not show up as pip installable for Py2. Making Murmur3 installation optional.
- v4-0-0: Ending Python 2 support, Adding more functionalities and documentation for DeepHash. Switching to Pytest for testing. Switching to Murmur3 128bit for hashing. Fixing classes which inherit from classes with slots didn't have all of their slots compared. Renaming ContentHash to DeepHash. Adding exclude by path and regex path to DeepHash. Adding ignore_type_in_groups. Adding match_string to DeepSearch. Adding Timedelta object diffing.
- v3-5-0: Exclude regex path
- v3-3-0: Searching for objects and class attributes
- v3-2-2: Adding help(deepdiff)
- v3-2-1: Fixing hash of None
- v3-2-0: Adding grep for search: object | grep(item)
- v3-1-3: Unicode vs. Bytes default fix
- v3-1-2: NotPresent Fix when item is added or removed.
- v3-1-1: Bug fix when item value is None (#58)
- v3-1-0: Serialization to/from json
- v3-0-0: Introducing Tree View
- v2-5-3: Bug fix on logging for content hash.
- v2-5-2: Bug fixes on content hash.
- v2-5-0: Adding ContentHash module to fix ignore_order once and for all.
- v2-1-0: Adding Deep Search. Now you can search for item in an object.
- v2-0-0: Exclusion patterns better coverage. Updating docs.
- v1-8-0: Exclusion patterns.

- v1-7-0: Deep Set comparison.
- v1-6-0: Unifying key names. i.e newvalue is new_value now. For backward compatibility, newvalue still works.
- v1-5-0: Fixing ignore order containers with unordered items. Adding significant digits when comparing decimals. Changes property is deprecated.
- v1-1-0: Changing Set, Dictionary and Object Attribute Add/Removal to be reported as Set instead of List. Adding Pypy compatibility.
- v1-0-2: Checking for ImmutableMapping type instead of dict
- v1-0-1: Better ignore order support
- v1-0-0: Restructuring output to make it more useful. This is NOT backward compatible.
- v0-6-1: Fixiing iterables with unhashable when order is ignored
- v0-6-0: Adding unicode support
- v0-5-9: Adding decimal support
- v0-5-8: Adding ignore order for unhashables support
- v0-5-7: Adding ignore order support
- v0-5-6: Adding slots support
- v0-5-5: Adding loop detection

CHAPTER

NINE

AUTHORS

- Sep Dehpour
  - Github
  - ZepWorks
  - Linkedin
  - Article about Deepdiff
- Victor Hahn Castell for major contributions
  - hahncastell.de
  - flexoptix.net
- nfvs for Travis-CI setup script.
- brbsix for initial Py3 porting.
- WangFenjin for unicode support.
- timoilya for comparing list of sets when ignoring order.
- Bernhard10 for significant digits comparison.
- b-jazz for PEP257 cleanup, Standardize on full names, fixing line endings.
- finnhughes for fixing __slots__
- moloney for Unicode vs. Bytes default
- serv-inc for adding help(deepdiff)
- movermeyer for updating docs
- maxrothman for search in inherited class attributes
- maxrothman for search for types/objects
- MartyHub for exclude regex paths
- sreecodeslayer for DeepSearch match_string
- Brian Maissy (brianmaissy) for weakref fix, enum tests
- Bartosz Borowik (boba-2) for Exclude types fix when ignoring order
- Brian Maissy (brianmaissy) for fixing classes which inherit from classes with slots didn't have all of their slots compared
- Juan Soler (Soleronline) for adding ignore_type_number
- mthaddon for adding timedelta diffing support

PYTHON MODULE INDEX

d

INDEX