Mitschreiben Documentation

Release 0.3 [4 - Beta]

sonntagsgesicht, based on a fork of Deutsche Postbank [pbrisk

Wednesday, 18 September 2019

Contents

CHAPTER 1

Introduction

1.1 Python project *mitschreiben*

mitschreiben (german for 'to take notes') helps recording values during calculations for later evaluation, e.g. check if the right objects or values were used or to present the results in structure of tables

It provides a class called Record which is basically used for everything. It grants access to record object, it is used for the recording and it is a context manager used to trigger whether to record or not.

1.2 Example Usage

In the first `Record(key = value)` or `Record(dictionary)` is placed where one wants to record a value. The decorator `Prefix` provided by this class is used to define a key extension under which the recorded value will be stored in the `Record`. The Prefixes get stacked, so when there is a successive function call to another function which is prefixed those Prefixes are concatenated.

```
from mitschreiben import Record

def magical_stuff_happens(baz, barz):
    return "That's", "great"

class Foo():

    @Record.Prefix()
    def bar(self, baz, barz)
        some_value1, some_value2 = self.do_something(baz, barz)

        Record(a_key=some_value1, another_key=some_value2)

        return some_value1, some_value2

    @Record.Prefix()
    def do_something(self, baz, barz):

        a_dict = {'again_a_key': baz, 'so_creative': barz}

        Record(a_dict)

        return magical_stuff_happens(baz, barz)

    def __repr__(self):
        return "Foo({})".format(id(self))
```

Now, since `Record` is a contextmanager, the recording will only happen in such a context. The `with`-statement activates the recording and returns the current scopes record object for convenient access. Another thing is, that record level is increased by this statement, leading to record objects that are only available in that scope. When leaving the `with` the outer scopes's record will be extend by the inner one, by prepending the outer records current prefix stack to each key of the inner one.

```
with Record() as rec:
    foo = Foo()
    foo.do_something("baz", "barz")
    foo.bar("baz","barz")

    print rec.entries
```

The entries are a dict whose keys are tuples which are the stacked Prefixes. In this way it is possible to determine which method on which object was called, what then led to successive calls, where in the end a value is recorded. The example above has the following output.

```
{('Foo(42403656).do_something', 'again_a_key'): 'baz', ('Foo(42403656).bar',
↪'Foo(42403656).do_something', 'again_a_key'): 'baz', ('Foo(42403656).do_something
↪', 'so_creative'): 'barz', ('Foo(42403656).bar', 'a_key'): "That's", (
↪'Foo(42403656).bar', 'another_key'): 'great', ('Foo(42403656).bar',
↪'Foo(42403656).do_something', 'so_creative'): 'barz'}
```

1.3 Formatting the output

The Record can be represented in different formats. The base to this is a *tree of dictionaries*, implemented by the class `DictTree` in `mitschreiben.formatting`. For the two base outputs however, one does not need to actually instantiate a `DictTree` yourself. The respective methods are

Both of these methods produce tables of the output. The idea is that, that certain calculations are made with different objects, leading to the same keywords. So one obtains a table with row keys (object names) and column keys (the keywords used to record a value). As the name of the former methods suggests, it produces this tables

and writes them as single .csv files into `Path`, whereas the latter construct a html document in which one can navigate through the tree structure and see the tables at those positions where they would be placed in the tree. Those tables would look similar to

```
<div class='panel-elem'><table>
<tr class='headrow'>
<th colspan='5'>table</th>
</tr>
<tr class='bodyrow'>
<th> </th>
<th>a_key</th>
<th>again_a_key</th>
<th>another_key</th>
<th>so_creative</th>
</tr>
<tr class='bodyrow'>
<th>Foo(42403656).bar</th>
<td>That's</td>
<td>None</td>
<td>great</td>
<td>None</td>
</tr><tr class='bodyrow'>
<th>Foo(42403656).do_something</th>
<td>None</td>
<td>baz</td>
<td>None</td>
<td>barz</td>
</tr></table></div>
<div class='panel'>
<div class='panel-elem'><table>
<tr class='headrow'>
<th colspan='2'>table</th>
</tr>
<tr class='bodyrow'>
<th> </th>
<th>Foo(42403656).do_something</th>
</tr>
<tr class='bodyrow'>
<th>again_a_key</th>
<td>baz</td>
</tr><tr class='bodyrow'>
<th>so_creative</th>
<td>barz</td>
</tr></table></div>
```

Another way would be to work with the `DictTree` directly.

```
from mitschreiben.formatting import DictTree

DT = DictTree(rec.entries)

tables = DT.make_tables()
for t in tables:
    print t.pretty_string()
    print
```

This results in the following output. The first table represents the top level of the record, whereas the other tabels are named by *object.function*.

```
                    Values |  a_key | again_a_key | another_key | so_creative
         Foo(42403656).bar | That's |         None |       great |         None
Foo(42403656).do_something |   None |          baz |        None |         barz
```

(continues on next page)

(continued from previous page)

```
Foo(42403656).bar
                    Values | again_a_key | so_creative
Foo(42403656).do_something |          baz |         barz
```

CHAPTER 2

Tutorial

. . . will come soon.

CHAPTER 3

API Documentation

3.1 Class List

mitschreiben.recording.Record	This class can be used to record values during calculations.
mitschreiben.table.Table	A table consist of columns and rows.
mitschreiben.formatting.DictTree	A class to work with a dict whose keys are tuples as if this dict was a dictionary of dictionaries of dictionaries...

3.2 Classes

class mitschreiben.recording.**Record**

Bases: object

This class can be used to record values during calculations. The class can be called to do the actual recording. Moreover the class grants access to the record depending on the record level and finally it is a contextmanager to control whether to record or not.

The call "Record()" yields the Record-Object of the current scope of recording. Within each scope this object is unique.

The call `Record(key=value,key2=value2,...)` or `Record(keyval) [with keyval={key:value, key2:value2, ...}]` records the values with the given keys in the Record-Object. The actual keys will be concatenated keys depending on a stack of prefixes (see below). The Record-Object knows this stack and records `{"former|keystack|key" : value}`.

By default no value will be recorded unless the recording is started. This is can be done in two ways.

1. Record is a contextmanager. A call looks like this:

```
Record(key=value)       # No recording of key, value
with Record() [as rec]:
    ...
Record(key=value)       # key, value is recorded
Record(key=value)       # No recording of key, value
```

(continues on next page)

(continued from previous page)

```
Thereby it is ensured to also stop the recording after the leaving the␣
↪with environment.
```

2. The call *Record().start()* enables recording whereas *Record().stop()* stops the recording

```
Record(key=value)       # No recording of key, value
Record().start()
Record(key=value)       # key, value is recorded
Record().stop()
Record(key=value)       # No recording of key, value
```

There can be different scopes of recording by using the context management functionality of Record. As soon as the recording context is entered the level of record is incremented by one and a within this scope this Record-Object does not know what might have been recorded in an outer scope. When leaving the inner scope the Record-Object will be integrated in the Record-Object of the outer scope.

With *Record().entries* one access the dict containing the recorded keys and values.

Record makes use of two subclasses: Key and Prefix

Key is class that provides some convenience to add keys and obtain new keys, which are used as the keys of the of the *Record().entries*.

The keys may contain some information on the call stack when the class Prefix is used. As mentioned above, Record knows a prefix stack two record in which successive function call a value was recorded. A Prefix is added when a function is decorated with @Record.Prefix().

input

```
@Record.Prefix()
def foo(obj):
    ...
Record(key=value)


with Record() as rec:
    foo(bar)

print rec.entries
```

output

```
{"bar.foo|key":value}
```

class Key
Bases: tuple

Key for the record entries

class Prefix(*prefix=None*)
Bases: object

This decorator generates a key extension depending on the method it decorates and the object that is passed to that method. This class remembers which methods have been decorated.

classmethod logged_methods()

classmethod autologging(*boolean*)

is_started
Returns a bool whether the Record is started or not. If False, *Record(*args, **kwargs)* has no effect.

entries
returns a dictionary with Recordkeys and Values

to_csv_files(*path*)
creates csv files for the different levels of the record in the given path.

to_html_tables(*filename*, *path=None*)
creates a html structured like the levels of the graph (directory like) where the last two branch levels are made into a table

clear()
this method clears the entries of a Record instance. Since there is only one toplevel Record instance everything is recorded there during its lifetime.

start()

stop()

append_prefix(*prefix*)
extend the current prefix stack by the prefix. If used as contextmanager the prefix will be removed outside of the context

pop_prefix()
remove the last extension from the prefix stack

class mitschreiben.table.**Table**(*default_value=None*, *name=None*, *left_upper=None*)
Bases: object

A table consist of columns and rows. Each entry has a row and a column key.

rows_count

cols_count

is_empty()

transpose()

append(*row_key*, *col_key*, *value*)

append_row(*row_key*, *row*)

get(*row_key*, *col_key*)

get_row(*row_key*)

get_row_list(*row_key*, *col_keys*)
returns the values of the row:row_key for all col_keys.

get_column(*col_key*)

get_default()

sort(*row_compare=None*, *column_compare=None*)

to_csv(*leftUpper=None*, *tabName=None*, *separator=';'*)

pretty_string(*leftUpper=None*, *tabName=None*, *separator=' | '*)

to_html()

to_nested_list()
returns the table as a nested list rows

static create_from_json_dict(*json_dict*)

class mitschreiben.formatting.**DictTree**
Bases: dict

A class to work with a dict whose keys are tuples as if this dict was a dictionary of dictionaries of dictionaries... When trying to look up a value with key (=tuple) and this tuple is partially contained in other keys (=tuples) than a DictTree with only those truncated keys is returned.

toplevel_tables(*name*)
Return tables from the two uppermost layers of the DictTree. One of them is a true table and the other is a collection of values

to_tables()
Makes a table from each level within the DictTree and returns those tables stored in a new DictTree

pretty_print()
this function prints an alphabetically sorted tree in a directory-like structure.

as_tree_to_html(*filename*, *path=None*)
This function creates a html file that presents the dicttree in its tree structure.

as_tables_to_html(*filename*, *path=None*)
This functions creates a html file presenting the tree in tables

as_html_tree_table(*filename*, *path=None*)
This function creates a html file, that is structured like a tree, where the last two-level-deep branches are represented as tables

to_csv_files(*path*)
this function creates csv files for every table that can be made from the tree

CHAPTER 4

Releases

These changes are listed in decreasing version number order.

4.1 Release 0.3

Release date was Wednesday, 18 September 2019

4.2 Release 0.2

December 31th, 2017

4.3 Release 0.1

Release date was July 7th, 2017

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index