Climate categories Documentation

Release 0.10.1

Mika Pflüger

Jan 25, 2024

CONTENTS:

CHAPTER

ONE

CLIMATE CATEGORIES

Commonly used codes, categories, terminologies, and nomenclatures used in climate policy analysis in a nice Python package. The documentation can be found at: https://climate-categories.readthedocs.io.

1.1 Included categorizations

Name	Title
IPCC1996	IPCC GHG emission categories (1996)
IPCC2006	IPCC GHG emission categories (2006)
IPCC2006_PRIMAP	IPCC GHG emission categories (2006) with additional categories
CRF1999	Common Reporting Format GHG emissions categories (1999)
CRF2013	Common Reporting Format GHG emissions categories (2013)
CRF2013_2021	CRF categories extended with country specific categories from 2021 submissions
CRF2013_2022	CRF categories extended with country specific categories from 2022 submissions
CRF2013_2023	CRF categories extended with country specific categories from 2023 submissions
BURDI	BUR GHG emission categories (DI query interface)
BURDI_class	BUR GHG emission categories (DI query interface) + classifications
CRFDI	CRF GHG emission categories (DI query interface)
CRFDI_class	CRF GHG emission categories (DI query interface) + classifications
GCB	Global Carbon Budget CO2 emission categories
RCMIP	RCMIP emissions categories
gas	Gases and other climate-forcing substances
ISO3	Countries, country groups, and other areas from ISO 3166
ISO3_GCAM	dito, plus regions used by the GCAM integrated assessment model

1.2 Included conversions between categorizations

- IPCC1996 <-> IPCC2006

1.3 Status

Climate categories is still in development and the API and names of categorizations are still subject to change.

1.4 License

1.5 Citation

If you use this library and want to cite it, please cite it as:

Mika Pflüger, Annika Günther, Johannes Gütschow, and Robert Gieseke. (2024-01-25). pik-primap/climate_categories: climate_categories Version 0.10.1. Zenodo. https://doi.org/10.5281/zenodo.10569044

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install Climate categories, run this command in your terminal:

```
$ pip install climate_categories
```

This is the preferred method to install Climate categories, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Climate categories can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/pik-primap/climate_categories
```

Or download the tarball:

```
$ curl -OJL https://github.com/pik-primap/climate_categories/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

THREE

USAGE

3.1 Categorizations

3.1.1 Included categorizations

In the `climate_categories` package, the categorizations are available directly at the top-level namespace, and as a dictionary in `.cats`:

[1]:
```
import climate_categories

climate_categories.cats
```

[1]:
```
{'IPCC1996': <Categorization IPCC1996 'IPCC GHG emission categories (1996)' with 233␣
↪categories>,
 'IPCC2006': <Categorization IPCC2006 'IPCC GHG emission categories (2006)' with 290␣
↪categories>,
 'IPCC2006_PRIMAP': <Categorization IPCC2006_PRIMAP 'IPCC GHG emission categories (2006)␣
↪with custom categories used in PRIMAP' with 303 categories>,
 'CRF1999': <Categorization CRF1999 'Common Reporting Format GHG emissions categories␣
↪(1999)' with 400 categories>,
 'CRF2013': <Categorization CRF2013 'Common Reporting Format GHG emissions categories␣
↪(2013)' with 729 categories>,
 'CRF2013_2021': <Categorization CRF2013_2021 'Common Reporting Format GHG emissions␣
↪categories (2013). Extended for 2021 CRF submissions.' with 960 categories>,
 'CRF2013_2022': <Categorization CRF2013_2022 'Common Reporting Format GHG emissions␣
↪categories (2013). Extended for 2022 CRF submissions.' with 966 categories>,
 'CRF2013_2023': <Categorization CRF2013_2023 'Common Reporting Format GHG emissions␣
↪categories (2013). Extended for 2023 CRF submissions.' with 967 categories>,
 'CRFDI': <Categorization CRFDI 'CRF GHG emission categories (DI query interface)' with␣
↪409 categories>,
 'CRFDI_class': <Categorization CRFDI_class 'CRF GHG emission categories (DI query␣
↪interface) + classifications' with 1133 categories>,
 'BURDI': <Categorization BURDI 'BUR GHG emission categories (DI query interface)' with␣
↪237 categories>,
 'BURDI_class': <Categorization BURDI_class 'BUR GHG emission categories (DI query␣
↪interface) + classifications' with 395 categories>,
 'GCB': <Categorization GCB 'Global Carbon Budget CO2 Emissions' with 9 categories>,
 'RCMIP': <Categorization RCMIP 'Emissions categories from the Reduced Complexity Model␣
↪Intercomparison Project (RCMIP)' with 87 categories>,
 'gas': <Categorization gas 'climate-forcing gases' with 242 categories>,
 'ISO3': <Categorization ISO3 'ISO 3166-1 countries with climate-relevant groupings'␣
```

(continues on next page)

(continued from previous page)

```
↪with 264 categories>,
 'ISO3_GCAM': <Categorization ISO3_GCAM 'ISO 3166-1 countries with climate-relevant␣
↪groupings with GCAM regions' with 456 categories>}
```

[2]:
```
climate_categories.IPCC2006
```

[2]:
```
<Categorization IPCC2006 'IPCC GHG emission categories (2006)' with 290 categories>
```

[3]:
```
climate_categories.cats["IPCC2006"]
```

[3]:
```
<Categorization IPCC2006 'IPCC GHG emission categories (2006)' with 290 categories>
```

Metadata for each categorization are accessible as properties:

[4]:
```
print(climate_categories.IPCC2006.name)
print(climate_categories.IPCC2006.title)
print(climate_categories.IPCC2006.comment)
print(climate_categories.IPCC2006.references)
print(climate_categories.IPCC2006.institution)
print(climate_categories.IPCC2006.last_update)
print(climate_categories.IPCC2006.version)
```

```
IPCC2006
IPCC GHG emission categories (2006)
IPCC classification of green-house gas emissions into categories, 2006 edition
IPCC 2006, 2006 IPCC Guidelines for National Greenhouse Gas Inventories, Prepared by the␣
↪National Greenhouse Gas Inventories Programme, Eggleston H.S., Buendia L., Miwa K.,␣
↪Ngara T. and Tanabe K. (eds). Volume 1, Chapter 8, Table 8.2, https://www.ipcc-nggip.
↪iges.or.jp/public/2006gl/vol1.html
IPCC
2010-06-30
2006
```

The categorization can be used as a dictionary mapping category codes to categories:

[5]:
```
climate_categories.IPCC2006["1.A"]
```

[5]:
```
<IPCC2006: '1.A'>
```

You can also query using alternative spellings of the code:

[6]:
```
climate_categories.IPCC2006["1A"]
```

[6]:
```
<IPCC2006: '1.A'>
```

For the categories, metadata is also available: a `title`, maybe a `comment`, all of its `codes` and possibly additional non-standard information in the `info` dictionary:

[7]:
```
one_a = climate_categories.IPCC2006["1.A"]
print(one_a.title)
print(one_a.comment)
print(one_a.codes)
print(one_a.info)
```

```
Fuel Combustion Activities
Emissions from the intentional oxidation of materials within an apparatus that is␣
↪designed to raise heat and provide it either as heat or as mechanical work to a␣
↪process or for use away from the apparatus.
('1.A', '1A')
{'gases': ['CO2', 'CH4', 'N2O', 'NOx', 'CO', 'NMVOC', 'SO2'], 'corresponding_categories_
↪IPCC1996': ['1A']}
```

For hierarchical categorizations, you can also query for parent and child categories. Note that a list of sets of children is returned in case a category can be composed differently:

[8]:
```
climate_categories.IPCC2006["1.A"].children
```

[8]:
```
[{<IPCC2006: '1.A.1'>,
  <IPCC2006: '1.A.2'>,
  <IPCC2006: '1.A.3'>,
  <IPCC2006: '1.A.4'>,
  <IPCC2006: '1.A.5'>}]
```

[9]:
```
climate_categories.IPCC2006["1.A"].parents
```

[9]:
```
{<IPCC2006: '1'>}
```

Finally, you can check if a categorization is hierarchical, and for hierarchical categorizations you can check if the sum of all child categories should be equal to the sum of parent categories:

[10]:
```
print(f"Hierachical: {climate_categories.IPCC2006.hierarchical}")
print(f"Total sum: {climate_categories.IPCC2006.total_sum}")
```

```
Hierachical: True
Total sum: True
```

3.1.2 Visualization

The relationships between categories in a hierarchical categorization can be visualized in a tree-like fashion:

[11]:
```
# Limit the maximum depth shown using `maxdepth`
print(climate_categories.IPCC2006.show_as_tree(maxdepth=2))
```

```
0 National Total
├1 Energy
├2 Industrial Processes and Product Use
├3 Agriculture, Forestry, and Other Land Use
├4 Waste
5 Other
```

[12]:
```
# Print only a part of tree using `root`
print(climate_categories.IPCC2006.show_as_tree(root="1A1"))
```

```
1.A.1 Energy Industries
├1.A.1.a Main Activity Electricity and Heat Production
│ ├1.A.1.a.i Electricity Generation
│ ├1.A.1.a.ii Combined Heat and Power Generation (CHP)
```

(continues on next page)

(continued from previous page)

```
│1.A.1.a.iii Heat Plants
├1.A.1.b Petroleum Refining
1.A.1.c Manufacture of Solid Fuels and Other Energy Industries
 ├1.A.1.c.i Manufacture of Solid Fuels
 1.A.1.c.ii Other Energy Industries
```

3.1.3 Child sets in hierarchical categorizations

For hierarchical categorizations, it is possible that a category can be composed of multiple child sets. As an example, in emissions reporting it is possible to report industrial emissions either by industry sectors, or by fuel. In this case, the parent `industry` category has two sets of children: either all the industry sectors, or all of the fuels.

You can see this with two toy example categorizations included in `climate_categories`:

[13]:
```
import climate_categories.tests.examples

HierEx = climate_categories.tests.examples.HierEx()
print(
    "Hierarchical categorization with only one way to subdivide the top category:"
)
print(HierEx.show_as_tree())

HierAltEx = climate_categories.tests.examples.HierAltEx()
print(
    "\nHierarchical categorization with two ways to subdivide the top category:"
)
print(HierAltEx.show_as_tree())
```

```
Hierarchical categorization with only one way to subdivide the top category:
0 Total
├1 Sector 1
├2 Sector 2
3 Sector 3


Hierarchical categorization with two ways to subdivide the top category:
0 Total
 ('0 Total's children, option 1)
├1 Sector 1
├2 Sector 2
3 Sector 3
 ('0 Total's children, option 2)
├a Fuel a
├b Fuel b
c Fuel c
```

As you can see, alternative ways to subdivide a category are indicated with double lines in `show_as_tree`. Programmatically, the difference is clear because `children` contains a list of possible child sets:

[14]:
```
HierEx["0"].children
```

[14]:
```
[{<HierEx: '1'>, <HierEx: '2'>, <HierEx: '3'>}]
```

[15]:
```
HierAltEx["0"].children
```

[15]:
```
[{<HierAltEx: '1'>, <HierAltEx: '2'>, <HierAltEx: '3'>},
 {<HierAltEx: 'a'>, <HierAltEx: 'b'>, <HierAltEx: 'c'>}]
```

3.1.4 Finding leaf descendants

For purposes like re-calculating top-level categories from simple leaf categories, it is useful to find the descendants of a category which have no children. Use the `leaf_children` property to do so.

[16]:
```
HierEx["0"].leaf_children
```

[16]:
```
[{<HierEx: '1'>, <HierEx: '2'>, <HierEx: '3'>}]
```

3.1.5 Extending categorizations

Often, you want to use a common categorization, but for one reason or another, you have to add a couple of categories. This is possible:

[17]:
```
IPCC2006_lulucf_extra = climate_categories.IPCC2006.extend(
    name="IPCC2006_lulucf_extra",
    categories={
        "M0.EL": {
            "title": "Total excluding lulucf",
            "comment": "All emissions and removals except emissions from land use, land␣
↪use change, and forestry",
        }
    },
    children=[("M0.EL", ("1", "2", "4", "5"))],
)

print(IPCC2006_lulucf_extra.name)
print(IPCC2006_lulucf_extra.title)
print(IPCC2006_lulucf_extra.comment)
print(IPCC2006_lulucf_extra.show_as_tree(maxdepth=2))
```

```
IPCC2006_IPCC2006_lulucf_extra
IPCC GHG emission categories (2006) + IPCC2006_lulucf_extra
IPCC classification of green-house gas emissions into categories, 2006 edition extended␣
↪by IPCC2006_lulucf_extra
0 National Total
├1 Energy
├2 Industrial Processes and Product Use
├3 Agriculture, Forestry, and Other Land Use
├4 Waste
5 Other

M0.EL Total excluding lulucf
```

(continues on next page)

(continued from previous page)

```
├1 Energy
├2 Industrial Processes and Product Use
├4 Waste
5 Other
```

If the canonical top level category of hierarchical categorizations is defined, you can also calculate the level of a category in the hierarchy:

[18]:
```
print(climate_categories.IPCC2006["0"].level)
print(climate_categories.IPCC2006["1.A"].level)
```

```
1
3
```

3.1.6 Pandas integration

For each categorization, the categories are also available as a pandas DataFrame:

[19]:
```
climate_categories.IPCC2006.df
```

[19]:
```
                                                  title  \
0                                        National Total
1                                                Energy
1.A                                Fuel Combustion Activities
1.A.1                                 Energy Industries
1.A.1.a      Main Activity Electricity and Heat Production
...                                                   ...
4.D.2        Industrial Wastewater Treatment and Discharge
4.E                                  Other (Please Specify)
5                                                   Other
5.A      Indirect N2O Emissions from The Atmospheric De...
5.B                                  Other (Please Specify)

                                                comment alternative_codes  \
0                                All emissions and removals                ()
1        This category includes all GHG emissions arisi...                ()
1.A      Emissions from the intentional oxidation of ma...            (1A,)
1.A.1    Comprises  emissions  from  fuels  combusted  ...           (1A1,)
1.A.1.a  Sum of emissions from main activity producers ...          (1A1a,)
...                                                   ...              ...
4.D.2    Treatment and discharge of liquid wastes and s...           (4D2,)
4.E      Release of GHGs from other waste handling acti...            (4E,)
5                                                   None                ()
5.A      Excluding indirect emissions from NOx and NH3 ...            (5A,)
5.B      Only use this category exceptionally, for any ...            (5B,)

                                      children
0                            ((1, 2, 3, 4, 5),)
1                            ((1.A, 1.B, 1.C),)
1.A       ((1.A.1, 1.A.2, 1.A.3, 1.A.4, 1.A.5),)
1.A.1             ((1.A.1.a, 1.A.1.b, 1.A.1.c),)
```

(continues on next page)

(continued from previous page)

```
1.A.1.a  ((1.A.1.a.i, 1.A.1.a.ii, 1.A.1.a.iii),)
...                                           ...
4.D.2                                          ()
4.E                                            ()
5                                     ((5.A, 5.B),)
5.A                                            ()
5.B                                            ()

[290 rows x 4 columns]
```

3.1.7 Finding unknown codes

Searching for a code in all included categorizations is possible using the `find_code` function:

[20]:
```
climate_categories.find_code("1A")
```

[20]:
```
{<IPCC2006_PRIMAP: '1.A'>,
 <IPCC2006: '1.A'>,
 <BURDI_class: '1.A'>,
 <BURDI: '1.A'>,
 <CRF2013_2023: '1.A'>,
 <CRF1999: '1.A'>,
 <CRF2013_2021: '1.A'>,
 <CRF2013_2022: '1.A'>,
 <CRF2013: '1.A'>,
 <IPCC1996: '1.A'>,
 <CRFDI: '1.AA'>,
 <CRFDI_class: '1.AA'>}
```

3.2 Conversions

3.2.1 Included conversions

You can get the rules for conversion between categories using the conversion objects:

[21]:
```
conv = climate_categories.IPCC1996.conversion_to("IPCC2006")
conv
```

[21]:
```
<Conversion 'IPCC1996' <-> 'IPCC2006' with 153 rules>
```

The conversion object can be queried for metadata:

[22]:
```
print(conv.categorization_a)
print(conv.categorization_b)
print(conv.auxiliary_categorizations)
print(conv.comment)
print(conv.institution)
print(conv.references)
print(conv.version)
print(conv.last_update)
```

```
IPCC1996
IPCC2006
[<Categorization gas 'climate-forcing gases' with 242 categories>]
None
IPCC
IPCC 2006, 2006 IPCC Guidelines for National Greenhouse Gas Inventories, Prepared by the␣
↪National Greenhouse Gas Inventories Programme, Eggleston H.S., Buendia L., Miwa K.,␣
↪Ngara T. and Tanabe K. (eds). Volume 1, Chapter 8d, Table 8.2  https://www.ipcc-nggip.
↪iges.or.jp/public/2006gl/vol1.html
None
2021-07-09 00:00:00
```

More importantly, the conversion object also holds the actual conversion rules:

[23]:
```
len(conv.rules)
```

[23]:
```
153
```

[24]:
```
conv.rules[0]
```

[24]:
```
ConversionRule(factors_categories_a={<IPCC1996: '0'>: 1}, factors_categories_b={
↪<IPCC2006: '0'>: 1}, auxiliary_categories={<Categorization gas 'climate-forcing gases'␣
↪with 242 categories>: set()}, comment='', csv_line_number=5, csv_original_text='0,,0,',
↪ cardinality_a='one', cardinality_b='one', is_restricted=False)
```

For the rules, the most important parts are the categories and factors for each side and the specification for which auxiliary categories the rule is valid:

[25]:
```
rule = conv.rules[0]
print(rule.factors_categories_a)
print(rule.factors_categories_b)
print(rule.auxiliary_categories)
```

```
{<IPCC1996: '0'>: 1}
{<IPCC2006: '0'>: 1}
{<Categorization gas 'climate-forcing gases' with 242 categories>: set()}
```

In this example, the category 1 of the IPCC1996 categorization equals exactly the category 1 of the IPCC2006 categorization (i.e. for both, the factor is unity) and the rule is valid for all gases (the set of gases is empty, meaning an unrestricted rule).

Often, you might be interested in rules concerning a specific set of categories, which you can also fetch:

[26]:
```
one_or_two_rules = conv.relevant_rules(
    {climate_categories.IPCC1996["1"], climate_categories.IPCC1996["2"]}
)

for rule in one_or_two_rules:
    print("###")
    print(rule.format_human_readable())
```

```
###
IPCC1996 1 Energy

IPCC2006 1 Energy
```

(continues on next page)

(continued from previous page)

```
###
IPCC1996 2 Industrial Processes
IPCC1996 3 Solvent and Other Product Use

IPCC2006 2 Industrial Processes and Product Use
```

Here, we used the `format_human_readable()` function to get a nicely formatted output. You can see that for this conversion, the category 1 maps cleanly between the two categorizations, but for category 2, you need to combine categories 2 and 3 from IPCC1996 to get category 2 from IPCC2006.

3.2.2 Finding inconsistencies

Dealing with large conversion rule sets can be daunting, so there are a few functions to help when checking conversions:

[27]:
```
# describe all of the conversion rules
# We only print the start of the description because it is looong
print(conv.describe_detailed()[:200] + "...")
```

```
# Mapping between IPCC1996 and IPCC2006

## Simple direct mappings

IPCC1996 0 National Total
IPCC2006 0 National Total

IPCC1996 1 Energy
IPCC2006 1 Energy

IPCC1996 1.A Fuel Combustion Activities
IP...
```

[28]:
```
# Find potential over counting problems
suspected_problems = conv.find_over_counting_problems()
for p in suspected_problems:
    print(p)
    print()
```

```
<IPCC1996: '6.A.1'> is possibly counted multiple times
involved leave groups categories: [[<IPCC2006: '4.A.1'>], [<IPCC2006: '4.A.3'>]]
involved rules: <Rule '0,,0,' from line 5>, <Rule '6,,4,' from line 146>, <Rule '6.A,,4.
↪A,' from line 147>, <Rule '6.A.1,,4.A.1,' from line 148>, <Rule '6.A.1 + 6.A.2,,4.A.3,
↪if possible prefer the individual rules above' from line 150>.

<IPCC1996: '6.A.2'> is possibly counted multiple times
involved leave groups categories: [[<IPCC2006: '4.A.2'>], [<IPCC2006: '4.A.3'>]]
involved rules: <Rule '0,,0,' from line 5>, <Rule '6,,4,' from line 146>, <Rule '6.A,,4.
↪A,' from line 147>, <Rule '6.A.2,,4.A.2,' from line 149>, <Rule '6.A.1 + 6.A.2,,4.A.3,
↪if possible prefer the individual rules above' from line 150>.

<IPCC1996: '6.A.3'> is possibly counted multiple times
```

(continues on next page)

(continued from previous page)

```
involved leave groups categories: [[<IPCC2006: '4.A'>], [<IPCC2006: '4.B'>]]
involved rules: <Rule '0,,0,' from line 5>, <Rule '6,,4,' from line 146>, <Rule '6.A,,4.
↪A,' from line 147>, <Rule '6.A.3,,4.B,' from line 151>.

<IPCC1996: '6.B.3'> is possibly counted multiple times
involved leave groups categories: [[<IPCC2006: '4.D'>], [<IPCC2006: '4.E'>]]
involved rules: <Rule '0,,0,' from line 5>, <Rule '6,,4,' from line 146>, <Rule '6.B,,4.
↪D,' from line 152>, <Rule '6.D + 6.B.3,,4.E,' from line 156>.

<IPCC2006: '2.B.7'> is possibly counted multiple times
involved leave groups categories: [[<IPCC1996: '2.A.3'>, <IPCC1996: '2.A.4'>], [
↪<IPCC1996: '2.B'>, <IPCC1996: '3.C'>]]
involved rules: <Rule '0,,0,' from line 5>, <Rule '2 + 3,,2,' from line 81>, <Rule '2.A.
↪3 + 2.A.4,,2.A.3 + 2.A.4 + 2.B.7,' from line 85>, <Rule '2.B + 3.C,,2.B,' from line 88>
↪.

<IPCC2006: '2.B.9'> is possibly counted multiple times
involved leave groups categories: [[<IPCC1996: '2.B'>, <IPCC1996: '3.C'>], [<IPCC1996:
↪'2.E'>]]
involved rules: <Rule '0,,0,' from line 5>, <Rule '2 + 3,,2,' from line 81>, <Rule '2.B␣
↪+ 3.C,,2.B,' from line 88>, <Rule '2.E,,2.B.9,' from line 103>.

<IPCC2006: '2.B.9.a'> is possibly counted multiple times
involved leave groups categories: [[<IPCC1996: '2.B'>, <IPCC1996: '3.C'>], [<IPCC1996:
↪'2.E.1'>]]
involved rules: <Rule '0,,0,' from line 5>, <Rule '2 + 3,,2,' from line 81>, <Rule '2.B␣
↪+ 3.C,,2.B,' from line 88>, <Rule '2.E,,2.B.9,' from line 103>, <Rule '2.E.1,,2.B.9.a,
↪' from line 104>.

<IPCC2006: '2.B.9.b'> is possibly counted multiple times
involved leave groups categories: [[<IPCC1996: '2.B'>, <IPCC1996: '3.C'>], [<IPCC1996:
↪'2.E.2'>]]
involved rules: <Rule '0,,0,' from line 5>, <Rule '2 + 3,,2,' from line 81>, <Rule '2.B␣
↪+ 3.C,,2.B,' from line 88>, <Rule '2.E,,2.B.9,' from line 103>, <Rule '2.E.2,,2.B.9.b,
↪' from line 105>.
```

Note that `find_over_counting_problems` at the moment can't reliably detect all over counting problems and also some suspected problems might be fine under closer examination. Use this function only to generate hints for possible problems.

[29]:
```
# Find unmapped categories
missing_1996, missing_2006 = conv.find_unmapped_categories()
# returns sets of missing categories
# unfortunately, this conversion is not very complete:
print(len(missing_1996))
print(len(missing_2006))
```

```
71
110
```

[30]:
```
# the sets just contain regular categories which were forgotten
next(iter(missing_1996))
```

```
[30]: <IPCC1996: '4.C.1.b.i'>
```

3.2.3 Viewing the full conversion

To view the full conversion, also consider directly loading the source CSVs from the `climate_categories/data/` folder in a spreadsheet program.

CHAPTER

FOUR

API

4.1 Module contents

Access to all categorizations is provided directly at the module level, using the names of categorizations. To access the example categorization *Excat*, simply use *climate_categories.Excat* .

class climate_categories.**Categorization**(*, *categories: dict[str, dict]*, *name: str*, *title: str*, *comment: str*, *references: str*, *institution: str*, *last_update: date*, *version: None | str = None*)

Bases: `object`

A single categorization system.

A categorization system comprises a set of categories, and their relationships as well as metadata describing the categorization system itself.

Use the categorization object like a dictionary, where codes can be translated to their meaning using `cat[code]` and all codes are available using `cat.keys()`. Metadata about the categorization is provided in attributes. If *pandas* is available, you can access a *pandas.DataFrame* with all category codes, and their meanings at `cat.df`.

name

The unique name/code

Type
str

references

Citable reference(s)

Type
str

title

A short, descriptive title for humans

Type
str

comment

Notes and explanations for humans

Type
str

institution

Where the categorization originates

Type
str

last_update

The date of the last change

Type
datetime.date

version

The version of the Categorization, if there are multiple versions

Type
str, optional

hierarchical

True if descendants and ancestors are defined

Type
bool

all_keys() → KeysView[str]

Iterate over all codes for all categories.

conversion_to(*other:* Categorization | *str*) → *Conversion*

Get conversion to other categorization.

If conversion rules for this conversion are not included, raises NotImplementedError.

property df: DataFrame

All category codes as a pandas dataframe.

extend(*, *categories: None | dict[str, dict] = None*, *alternative_codes: None | dict[str, str] = None*, *name: str*, *title: None | str = None*, *comment: None | str = None*, *last_update: None | date = None*) → CategorizationT

Extend the categorization with additional categories, yielding a new categorization.

Metadata: the `name`, `title`, `comment`, and `last_update` are updated automatically (see below), the `institution` and `references` are deleted and the values for `version` and `hierarchical` are kept. You can set more accurate metadata (for example, your institution) on the returned object if needed.

Parameters

- **categories** (*dict, optional*) – Map of new category codes to their specification. The specification is a dictionary with the keys "title", optionally "comment", and optionally "alternative_codes".
- **alternative_codes** (*dict, optional*) – Map of new alternative codes. A dictionary with the new alternative code as key and existing code as value.
- **name** (*str*) – The name of your extension. The returned Categorization will have a name of "{old_name}_{name}", indicating that it is an extension of the underlying Categorization.
- **title** (*str, optional*) – A string to add to the original title. If not provided, " + {name}" will be used.
- **comment** (*str, optional*) – A string to add to the original comment. If not provided, " extended by {name}" will be used.

- **last_update** (*datetime.date, optional*) – The date of the last update to this extension. Today will be used if not provided.

Returns
Extended categorization

Return type
Categorization

static from_pickle(*filepath: str | Path | IO[bytes]*) → CategorizationT

De-serialize Categorization from a file written by to_pickle.

Note that this uses the pickle module, which executes arbitrary code in the provided file. Only load from pickle files that you trust.

static from_python(*filepath: str | Path | IO[bytes]*) → CategorizationT

De-serialize Categorization from a file written by to_python.

Note that this executes the python cache file. Only load from python cache files you trust.

classmethod from_spec(*spec: dict[str, Any]*) → CategorizationT

Create Categorization from a Dictionary specification.

classmethod from_yaml(*filepath: str | Path | TextIO*) → CategorizationT

Read Categorization from a StrictYaml file.

items() → ItemsView[str, *Category*]

Iterate over (primary code, category) pairs.

keys() → KeysView[str]

Iterate over the codes for all categories.

to_pickle(*filepath: str | Path*) → None

Serialize to a file using python's pickle.

to_python(*filepath: str | Path*) → None

Write spec to a Python file.

to_spec() → dict[str, Any]

Turn this categorization into a specification dictionary ready to be written to a yaml file.

Returns
spec – Specification dictionary understood by *from_spec*.

Return type
dict

to_yaml(*filepath: str | Path*) → None

Write to a YAML file.

values() → ValuesView[*Category*]

Iterate over the categories.

class climate_categories.**Category**(*codes: tuple[str, ...]*, *categorization:* Categorization, *title: str*, *comment: None | str = None*, *info: None | dict = None*)

Bases: object

A single category.

`to_spec()` → tuple[str, dict[str, str | dict | list]]

Turn this category into a specification ready to be written to a yaml file.

Returns
(code – Primary code and specification dict

Return type
str, spec: dict)

`class climate_categories.Conversion(*, categorization_a: Categorization, categorization_b: Categorization, rules: list[ConversionRule], auxiliary_categorizations: list[Categorization] | None = None, comment: str | None = None, references: str | None = None, institution: str | None = None, last_update: date | None = None, version: str | None = None)`

Bases: `ConversionBase`

Conversion between two categorizations.

This class collects functionality which needs access to the actual categorizations and categories.

`categorization_a`

The first categorization.

Type
Categorization

`categorization_b`

The second categorization.

Type
Categorization

`auxiliary_categorizations`

The auxiliary categorizations, if any.

Type
list of *Categorization*, optional

`rules`

The actual rules for conversion between individual categories or sets of categories.

Type
list of *ConversionRule*

`comment`

Notes and explanations for humans.

Type
str, optional

`references`

Citable reference(s) for the conversion.

Type
str, optional

`institution`

Where the conversion originates.

Type
str, optional

`last_update`

The date of the last change.

Type
datetime.date, optional

`version`

The version of the ConversionRules, if there are multiple versions.

Type
str, optional

`describe_detailed()` → str

Detailed human-readable description of the conversion rules.

Sections are added for direct one-to-one mappings, one-to-many mappings, many-to-one mappings, and many-to-many mappings, respectively.

Factors are shown at the start of the line if they don't equal 1, like this: -1 * IPCC1996 4 Agriculture to indicate that category 4 should be subtracted.

`find_over_counting_problems()` → list[OverCountingProblem]

Check if any category from one side is counted more than once on the other side.

Note that the algorithm at the moment can't reliably detect all over counting problems and also some suspected problems might be fine under closer examination, so use this function only to generate hints for possible problems.

Returns
problems – All detected suspected problems.

Return type
list of OverCountingProblem objects

`find_unmapped_categories()` → tuple[set[*Category*], set[*Category*]]

Find categories for which no rule exists to map them.

Returns
missing_categories_a, missing_categories_b – A list of categories missing from categorization_a and categorization_b, respectively.

Return type
set, set

`relevant_rules`(*categories: set[*HierarchicalCategory*]*, *source_categorization:* Categorization | *None* = *None*, *simple_sums_only: bool = False*) → list[*ConversionRule*]

Returns all rules which involve the given categories.

Parameters

- **categories** (*set of* HierarchicalCategory) – The categories to limit the rules to.
- **source_categorization** (Categorization, *optional*) – The categorization that the categories are part of, either self.categorization_a or self.categorization_b.
- **simple_sums_only** (*bool, default False*) – If true, only consider rules where the given categories enter as simple summands (i.e. with a factor of 1).

Returns
All rules which touch the given categories.

Return type
relevant_rules

`reversed`() → *Conversion*

Returns the Conversion with categorization_a and categorization_b swapped.

`class climate_categories.ConversionRule`(*factors_categories_a: dict[Category, int]*, *factors_categories_b: dict[Category, int]*, *auxiliary_categories: dict[Categorization, set[Category]]*, *comment: str = ''*, *csv_line_number: int | None = None*, *csv_original_text: str | None = None*)

Bases: `object`

A rule to convert between categories from two different categorizations.

Supports one-to-one relationships, one-to-many relationships in both directions and many-to-many relationships. For each category, a factor is given which can also be negative to model relationships like A = B - C.

Using auxiliary_categories, a rule can be restricted to specific auxiliary categories only.

`factors_categories_a`

Map of categories from the first categorization to factors. For a simple addition, use factor 1, to subtract the category, use factor -1.

Type
dict mapping categories to factors

`factors_categories_b`

Map of categories from the second categorization to factors. For a simple addition, use factor 1, to subtract the category, use factor -1.

Type
dict mapping categories to factors

`auxiliary_categories`

Map of auxiliary categorizations to sets of auxiliary categories. Not all auxiliary categorizations need to be specified, and if an auxiliary categorization is not specified (or an empty set of category codes is given), the validity of the rule is not restricted. If an auxiliary categorization is specified and categories are given, the rule is only valid for the given categories. If multiple auxiliary categorizations are given, the rule is only valid if all auxiliary categorizations match.

Type
dict[*Categorization*, set[*Category*]]

`comment`

A human-readable comment explaining the rule or adding additional information.

Type
str

`cardinality_a`

The cardinality of the rule on side a. Is “one” if there is exactly one category in factors_categories_a, and “many” otherwise.

Type
str

`cardinality_b`

The cardinality of the rule on side b. Is “one” if there is exactly one category in factors_categories_b, and “many” otherwise.

Type
str

`is_restricted`

The rule is restricted if and only if for at least one auxiliary categorization at least one category is specified, so that the rule is only valid for a subset of cases. Otherwise, the rule is unrestricted and valid for all cases.

Type
bool

`format_human_readable`(*categorization_separator: str = '\n'*) → str

Format the rule for humans.

Parameters
categorization_separator (*str, optional*) – The categorization_separator is printed between the categories from the source categorization and the categories from the target categorization to make the difference clear.

Returns
human_readable – The rule in a format optimized for error-free parsing by humans.

Return type
str

`format_with_lineno`() → str

Human-readable string representation of the rule with information in which line in the CSV file it was defined, if that is available.

`reversed`() → *ConversionRule*

Return the ConversionRule with categorization_a and categorization_b swapped.

`to_spec`() → ConversionRuleSpec

Return a serializable specification.

Returns
spec

Return type
ConversionRuleSpec

`class climate_categories.HierarchicalCategorization`(*, *categories: dict[str, dict]*, *name: str*, *title: str*, *comment: str*, *references: str*, *institution: str*, *last_update: date*, *version: None | str = None*, *total_sum: bool*, *canonical_top_level_category: None | str = None*)

Bases: *Categorization*

In a hierarchical categorization, descendants and ancestors (parents and children) are defined for each category.

`total_sum`

If the sum of the values of children equals the value of the parent for extensive quantities. For example, a Categorization containing the Countries in the EU and the EU could set *total_sum = True*, because the emissions of all parts of the EU must equal the emissions of the EU. On the contrary, a categorization of Industries with categories *Power:Fossil Fuels* and *Power:Gas* which are both children of *Power* must set *total_sum = False* to avoid double counting of fossil gas.

Type
bool

canonical_top_level_category

The level of a category is calculated with respect to the canonical top level category. Commonly, this will be the world total or a similar category. If the canonical top level category is not set (i.e. is `None`), levels are not defined for categories.

Type
HierarchicalCategory

ancestors(*cat: str | HierarchicalCategory*) → set[*HierarchicalCategory*]

All ancestors of the given category, i.e. the direct parents and their parents, etc.

children(*cat: str | HierarchicalCategory*) → list[set[*HierarchicalCategory*]]

The list of sets of direct children of the given category.

descendants(*cat: str | HierarchicalCategory*) → set[*HierarchicalCategory*]

All descendants of the given category, i.e. the direct children and their children, etc.

property df: DataFrame

All category codes as a pandas dataframe.

extend(*, *categories: None | dict[str, dict] = None*, *alternative_codes: None | dict[str, str] = None*, *children: None | list[tuple] = None*, *name: str*, *title: None | str = None*, *comment: None | str = None*, *last_update: None | date = None*) → *HierarchicalCategorization*

Extend the categorization with additional categories and relationships, yielding a new categorization.

Metadata: the `name`, `title`, `comment`, and `last_update` are updated automatically (see below), the `institution` and `references` are deleted and the values for `version`, `hierarchical`, `total_sum`, and `canonical_top_level_category` are kept. You can set more accurate metadata (for example, your institution) on the returned object if needed.

Parameters

- **categories** (*dict, optional*) – Map of new category codes to their specification. The specification is a dictionary with the keys "title", optionally "comment", and optionally "alternative_codes".
- **alternative_codes** (*dict, optional*) – Map of new alternative codes. A dictionary with the new alternative code as key and existing code as value.
- **children** (*list, optional*) – List of `(parent, (child1, child2, ...))` pairs. The given relationships will be inserted in the extended categorization.
- **name** (*str*) – The name of your extension. The returned Categorization will have a name of "{old_name}_{name}", indicating that it is an extension of the underlying Categorization.
- **title** (*str, optional*) – A string to add to the original title. If not provided, " + {name}" will be used.
- **comment** (*str, optional*) – A string to add to the original comment. If not provided, " extended by {name}" will be used.
- **last_update** (*datetime.date, optional*) – The date of the last update to this extension. Today will be used if not provided.

Returns
Extended categorization

Return type
HierarchicalCategorization

classmethod from_spec(*spec: dict[str, Any]*) → CategorizationT

Create Categorization from a Dictionary specification.

is_leaf(*cat: str | HierarchicalCategory*) → bool

Is the category a leaf category, i.e. without children?

items() → ItemsView[str, *HierarchicalCategory*]

Iterate over (primary code, category) pairs.

leaf_children(*cat: str | HierarchicalCategory*) → list[set[*HierarchicalCategory*]]

The sets of subcategories which are descendants of the category and do not have children themselves.

Sets of children are chased separately, so each set of leaf children is self-sufficient to reconstruct this category (if the categorization allows reconstructing categories from their children, i.e. if total_sum is set).

level(*cat: str | HierarchicalCategory*) → int

The level of the given category.

The canonical top-level category has level 1 and its children have level 2 etc.

To calculate the level, first only the first ("canonical") set of children is considered. Only if no path from the canonical top-level category to the given category can be found all other sets of children are considered to calculate the level.

parents(*cat: str | HierarchicalCategory*) → set[*HierarchicalCategory*]

The direct parents of the given category.

show_as_tree(**, format_func: ~typing.Callable[[~climate_categories._categories.HierarchicalCategory], str] = <class 'str'>, maxdepth: None | int = None, root: None | ~climate_categories._categories.HierarchicalCategory | str = None*) → str

Format the hierarchy as a tree.

Starting from the given root, or - if no root is given - the top-level categories (i.e. categories without parents), the tree of categories that are transitive children of the root is show, with children connected to their parents using lines. If a parent category has one set of children, the children are connected to each other and the parent with a simple line. If a parent category has multiple sets of children, the sets are connected to parent with double lines and the children in a set are connected to each other with simple lines.

Parameters

- **format_func** (*callable, optional*) – Function to call to format categories for display. Each category is formatted for display using format_func(category), so format_func should return a string without line breaks, otherwise the tree will look weird. By default, str() is used, so that the first code and the title of the category are used.
- **maxdepth** (*int, optional*) – Maximum depth to show in the tree. By default, goes to arbitrary depth.
- **root** (HierarchicalCategory *or str, optional*) – HierarchicalCategory object or code to use as the top-most category. If not given, the whole tree is shown, starting from all categories without parents.

Returns

tree_str – Representation of the hierarchy as formatted string. print() it for optimal viewing.

Return type

str

to_spec() → dict[str, Any]

Turn this categorization into a specification dictionary ready to be written to a yaml file.

Returns
spec – Specification dictionary understood by *from_spec*.

Return type
dict

`values()` → ValuesView[*HierarchicalCategory*]

Iterate over the categories.

`class climate_categories.HierarchicalCategory`(*codes: tuple[str]*, *categorization:* HierarchicalCategorization, *title: str*, *comment: None | str = None*, *info: None | dict = None*)

Bases: *Category*

A single category from a HierarchicalCategorization.

`property ancestors: set[HierarchicalCategory]`

The super-categories where this category or any of its parents is a member of any set of children, transitively.

Note that all possible ancestors are returned, not only "canonical" ones.

`property children: list[set[HierarchicalCategory]]`

The sets of subcategories comprising this category.

The first set is canonical, the other sets are alternative. Only the canonical sets are used to calculate the level of a category.

`property descendants: set[HierarchicalCategory]`

The sets of subcategories comprising this category directly or indirectly.

Note that all possible descendants are returned, not only "canonical" ones.

`property is_leaf: bool`

Is this category a leaf category, i.e. without children?

`property leaf_children: list[set[HierarchicalCategory]]`

The sets of subcategories which are descendants of this category and do not have children themselves.

Sets of children are chased separately, so each set of leaf children is self-sufficient to reconstruct this category (if the categorization allows reconstructing categories from their children, i.e. if total_sum is set).

`property level: int`

The level of the category.

The canonical top-level category has level 1 and its children have level 2 etc.

To calculate the level, only the first ("canonical") set of children is considered for intermediate categories.

`property parents: set[HierarchicalCategory]`

The super-categories where this category is a member of any set of children.

Note that all possible parents are returned, not "canonical" parents.

`to_spec()` → tuple[str, dict[str, str | dict | list]]

Turn this category into a specification ready to be written to a yaml file.

Returns
(code – Primary code and specification dict

Return type
str, spec: dict)

climate_categories.**find_code**(*code: str*) → set[*Category*]

Search for the given code in all included categorizations.

climate_categories.**from_pickle**(*filepath: str | Path | IO[bytes]*) → *Categorization* | *HierarchicalCategorization*

De-serialize Categorization or HierarchicalCategorization from a file written by to_pickle.

Note that this uses the pickle module, which executes arbitrary code in the provided file. Only load from pickle files that you trust.

climate_categories.**from_python**(*filepath: str | Path | IO[bytes]*) → CategorizationT

Read Categorization or HierarchicalCategorization from a python cache file.

Note that this executes the python cache file. Only load from python cache files you trust.

climate_categories.**from_spec**(*spec: dict[str, Any]*) → CategorizationT

Create Categorization or HierarchicalCategorization from a dict specification.

climate_categories.**from_yaml**(*filepath: str | Path | TextIO*) → CategorizationT

Read Categorization or HierarchicalCategorization from a StrictYaml file.

4.1.1 climate_categories.search module

climate_categories.search.**search_code**(*code: str*, *cats: Iterable[*Categorization*]*) → set[*Category*]

Search for the given code in the given categorizations.

CHAPTER

FIVE

DATA

5.1 Categorizations

The categorizations included in this package are stored as YAML 1.2 files. The data files must only use the subset of YAML's features understood by StrictYaml. The allowed contents are defined here.

5.1.1 Simple Categorizations

Non-hierarchical categorizations are stored in StrictYaml files with the following fields:

Key	Type	Notes	Example
name	str	a valid python variable name	IPCC2006
title	str	one-line description	IPCC GHG emission categories (2006)
comment	str	long-form description	IPCC classification of green-house…
references	str	citable reference(s) and sources	IPCC 2006, 2006 IPCC Guidelines…
institution	str	where the categorization is from	IPCC
last_update	str	date of last change in ISO format	2010-06-30
hierarchical	bool	has to be `no`, `false`, or `False`	no
version	str	optional	2006
categories	map	see below	

The metadata attributes are inspired by the CF conventions for the description of file contents.

The categories are given as a map from the primary code of the category to a dictionary specification with the following fields:

Key	Type	Notes	Example
title	str	one-line description of the category	Energy
comment	str	optional, long-form description	Includes all GHG…
alternative_codes	list	optional, alias codes	['1A', '1 A']
info	map	optional, arbitrary metadata	{'gases': ['CO2', 'NH3']}

The examples in the table are given in python syntax. An example in YAML syntax would be:

```
categories:
  '1':
    title: ENERGY
    comment: This category includes all GHG emissions arising from combustion and
```

(continues on next page)

(continued from previous page)

```
    fugitive releases of fuels. Emissions from the non-energy uses of fuels are
    generally not included here, but reported under Industrial Processes and Product
    Use Sector.
  info:
    gases:
    - CO2
    - CH4
1.A:
  title: Fuel Combustion Activities
  comment: Emissions from the intentional oxidation of materials within an apparatus
    that is designed to raise heat and provide it either as heat or as mechanical
    work to a process or for use away from the apparatus.
  alternative_codes:
  - 1A
  info:
    gases:
    - CO
    - NMVOC
    corresponding_categories_IPCC1996:
    - 1A
```

5.1.2 Hierarchical Categorizations

Hierarchical categorizations are also stored in StrictYaml files, with additional meta data fields:

Key	Type	Notes	Example
hierarchical	str	has to be `yes`, `true`, or `True`	yes
total_sum	bool	if parents are the sum of their children	True
canonical_top_level_category	str	optional, code of the highest category	TOTAL

In the category specifications, an additional optional key `children` is introduced which contains lists of lists of codes of children. Since some categories can be composed of different sets of children, it is necessary to give a list of lists.

An example in StrictYaml syntax with two categories would be:

```
categories:
  '1':
    title: ENERGY
    comment: This category includes all GHG emissions arising from combustion and
      fugitive releases of fuels. Emissions from the non-energy uses of fuels are
      generally not included here, but reported under Industrial Processes and Product
      Use Sector.
    info:
      gases:
      - CO2
      - CH4
    children:
      - - 1.A
        - 1.B
  1.A:
```

(continues on next page)

(continued from previous page)

```
  title: Fuel Combustion Activities
  comment: Emissions from the intentional oxidation of materials within an apparatus
    that is designed to raise heat and provide it either as heat or as mechanical
    work to a process or for use away from the apparatus.
  alternative_codes:
  - 1A
  info:
    gases:
    - CO
    - NMVOC
    corresponding_categories_IPCC1996:
    - 1A
```

5.2 Conversions

Conversion rules between categorizations included in this package are stored in comma separated value files with a tightly specified format. Commas separate fields, and can be escaped using a backslash. Two consecutive backslashes are read as a single backslash.

The file consists of a YAML meta data block at the start of the file, and a data block following it. The YAML meta data block lines start with the comment char '#'.

5.2.1 Meta data block

The meta data bloc consists of key-value pairs, one on each line, key and value separated by a colon. All meta data are optional, the allowed keys are:

Key	Type	Notes	Example
comment	str	Notes and explanations for humans.	Rules for agriculture are still missing.
references	str	Citable reference(s) for the conversion.	doi:10.00000/00
institution	str	Where the conversion originates.	PIK
last_update	date	ISO format of the date of the last change.	1999-12-31
version	str	The version, if there are multiple.	2.3

5.2.2 Data block

The data block starts with a header which defines the data columns. The first column must be the name of the first categorization, the following columns are the names of auxiliary categorizations, the penultimate column must be the name of the second categorization, and the last column must be `comment`.

After the header, any number of rules follow, with one rule per line. Each rule consists of:

- A formula for the first and second categorizations each, defining which categories are converted into each other.
- A list of categories for each auxiliary categorization, limiting the validity of the rule to specific categorizations.

The formulas contain sums and differences of category codes. The lists of auxiliary category codes are separated by whitespace. In formulas and lists, category codes that consist of alphanumeric characters and dots can be written directly, and other category codes must be enclosed in " characters.

5.2.3 Example

```
# comment: an example conversion
# institution: PIK
IPCC1996,gas,IPCC2006,comment
4 + 5,,3,Both sectors were combined
4.D,N2O,3.C.4 + 3.C.5,N2O emissions are separated into own categories
```

CHAPTER

SIX

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:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/pik-primap/climate_categories/issues.

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.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

6.1.3 New categorizations

Especially welcome are new categorizations, which are not included in climate_categories so far. Pull requests and issue reports at github are very welcome!

The categorizations are read from StrictYaml files located at `climate_categories/data/`. You can write a yaml definition by hand, but ideally, categorizations are generated from some canonical source automatically, so that the generation is reproducible and transparent. Scripts to generate categorizations are located in the `data_generation` folder and write their results directly to `climate_categories/data/`. For each data file, a target should be included in the top-level Makefile. Do *not* include source pdfs with non-free copyright licenses into the git repository. Instead, download them in the data generation scripts (see `data_generation/IPCC2006.py` for an example how to do that efficiently with caching).

Because all Categorizations are read in when importing `climate_categories` and parsing StrictYaml files is not very efficient, the categories should be also stored as cached Python files using the `to_python` instance method. Run *make cache* to generate these from the YAML files.

6.1.4 New conversions

Especially welcome as well are new conversions between categorizations, which are not included in climate_categories so far. Pull requests and issue reports at github are very welcome!

The conversions are read from CSV files located at `climate_categories/data/`. You can write a CSV definition by hand, but ideally, conversions are also generated from some canonical source automatically, so that the generation is reproducible and transparent. As the scripts to generate categorizations, the scripts to generate conversion files are located in the `data_generation` folder and write their results directly to `climate_categories/data/`.

Conversion files are read on demand and therefore no pickle files need to be generated.

6.1.5 Write Documentation

Climate categories could always use more documentation, whether as part of the official Climate categories docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.6 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/pik-primap/climate_categories/issues.

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 contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *climate_categories* for local development.

1. Fork the *climate_categories* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/climate_categories.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ cd climate_categories/
$ make virtual-environment
$ make install-pre-commit
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass our tests and automatically format everything according to our rules:

```
$ make lint
```

Often, the linters can fix errors themselves, so if you get failures, run `make lint` again to see if any errors need human intervention.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring and check the generated API documentation.
3. The pull request will be tested on python 3.9, 3.10, and 3.11.

6.4 Deploying

A reminder for the maintainers on how to deploy.

1. Commit all your changes.
2. Run `tbump X.Y.Z`.
3. Wait a bit that the release on github and zenodo is created.
4. Run `make README.rst` to update the citation information in the README from the zenodo API. Check if the version is actually correct, otherwise grab a tea and wait a little more for zenodo to mint the new version. Once it worked, commit the change.
5. Upload the release to pyPI: `make release`

CHAPTER

SEVEN

CREDITS

7.1 Developers

- Mika Pflüger <mika.pflueger@climate-resource.com>
- Annika Günther <annika.guenther@pik-potsdam.de>
- Johannes Gütschow <johannes.guetschow@pik-potsdam.de>
- Robert Gieseke <rob.g@web.de>

7.2 Libraries

This package was originally created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER

EIGHT

CHANGELOG

8.1 0.10.1 (2024-01-25)

- ISO3_GCAM: Removed extraneous “v” from version specifications in region codes

8.2 0.10.0 (2024-01-25)

- Added ISO3_GCAM categorization which contains regions used in the integrated assessment model GCAM.

8.3 0.9.2 (2023-06-22)

- ISO3: Add all parties to the UNFCCC as direct children of UNFCCC as first set of children. That way, it is easy to ergonomically get all parties to the UNFCCC without adding up Annex-I and Non-Annex-I parties manually.

8.4 0.9.1 (2023-06-15)

- Add AOSIS country group to ISO3 categorization.

8.5 0.9.0 (2023-06-14)

- Add ISO3 terminology for countries, areas, and country groups including UNFCCC signatories and Annex-I and Non-Annex-I groups and the evolution of the EU over time.

8.6 0.8.5 (2023-05-23)

- Re-release again.

8.7 0.8.4 (2023-05-23)

- Re-release to make sure py.typed is included in built package.

8.8 0.8.3 (2023-05-23)

- add py.typed file to announce this library is using type hints.

8.9 0.8.2 (2023-05-15)

- Remove pygments-csv-lexer dependency for docs building.
- Add function to find leaf children of a category, useful for re-calculating top-level categories from constituents.

8.10 0.8.1 (2023-04-26)

- regenerate data included in the package to benefit from latest fixes in data generation scripts.

8.11 0.8.0 (2023-04-26)

- Add updated CRF2013 terminologies for 2021, 2022, and 2023 submission rounds
- The unfccc DI API recently returns unspecified measure IDs. data_generation/CRFDI_class.py was fixed to ignore them.
- Add CRF2013 terminology for data submitted by AnnexI countries to the UNFCCC
- Drop support for Python 3.7 and 3.8, add support for Python 3.11

8.12 0.7.1 (2021-11-25)

- Change conversion metadata format to use comment chars and a YAML header.

8.13 0.7.0 (2021-11-25)

- Use Python files instead of pickle objects for caching

8.14 0.6.3 (2021-11-05)

- Export Category and HierarchicalCategory types.
- Add ConversionRule.is_restricted attribute to easily check if a rule is restricted to specific auxiliary categories.

8.15 0.6.2 (2021-11-05)

- Export Conversion and ConversionRule types.

8.16 0.6.1 (2021-11-04)

- Add emissions categorization from the Reduced Complexity Model Intercomparison Project (RCMIP). Thanks to Robert Gieseke for the contribution and Zeb Nicholls for input.

8.17 0.6.0 (2021-10-22)

- Automate changelog generation from snippets - avoids resolving merge conflicts manually
- Automate github releases.
- Add category "0" (National total) to IPCC1996 and IPCC2006 categorizations. While it is not in the official specification, it is widely used and adding it also enables automatically assigning a level to all other categories.
- Add categorization CRF1999 used within in the common reporting framework data.
- Refactor rendering of large categorizations using `show_as_tree()`, adding more clarity to alternative child sets. Add usage documentation for `show_as_tree()`. Thanks to Robert Gieseke for feedback.
- Fixes for IPCC2006 categorization (and IPCC2006_PRIMAP):
  - proper title for category 3.B.3.a "Grassland Remaining Grassland"
  - correct corresponding 1996 category for category 1.A.4.c.ii
- Fixes for IPCC1996 categorization:
  - category 4.B.10 has the correct title "Anaerobic Lagoons"
  - correct usage of units in the titles of categories 4.C.3.a and 4.C.3.b
- Add mechanism to describe conversions between categorizations.
- Add conversion between IPCC2006 and IPCC1996.
- Add algorithm to detect over counting in conversions between categorizations.
- Refactor generation of IPCC2006 and IPCC1996 categorizations.
- Add function to find unmapped categories in a conversion.

8.18 0.5.4 (2021-10-18)

- Add Global Carbon Budget categorization.

8.19 0.5.3 (2021-10-12)

- Add gas categorization which includes commonly used climate forcing substances.

8.20 0.5.2 (2021-05-18)

- Add IPCC2006_PRIMAP categorization.
- Add refrigerant sub-classes and additional codes to CRFDI_class.

8.21 0.5.1 (2021-05-04)

- Add BURDI, CRFDI, BURDI_class, and CRFDI_class categorizations and scripts to generate them from the UNFCCC DI flexible query API.

8.22 0.5.0 (2021-03-23)

- Switch `to_yaml()` output to ruamel.yaml so that valid, correctly typed YAML 1.2 is written. This should enable easier re-use of the data in other contexts.
- Consistently use title case for titles in IPCC categorizations.

8.23 0.4.0 (2021-03-17)

- Add more unit tests.
- Add consistency tests for IPCC categorizations.
- Update documentation.
- Add data format documentation.

8.24 0.3.2 (2021-03-16)

- Use tbump for simpler versioning.

8.25 0.3.1 (2021-03-16)

- Properly include data files in binary releases.

8.26 0.3.0 (2021-03-16)

- Add IPCC1996 categorization and scripts to generate it from the source pdf.
- Change packaging to declarative style.
- Automate generation of pickled files via Makefile.
- Automate loading of included categorizations.

8.27 0.2.2 (2021-03-09)

- Re-release again to trigger zenodo.

8.28 0.2.1 (2021-03-09)

- Re-release to include correct changelog.

8.29 0.2.0 (2021-03-09)

- Introduce API for multiple codes and multiple children.
- Implement classes and functions.
- Add IPCC2006 categorization and scripts to generate it from the source pdf.

8.30 0.1.0 (2021-01-18)

- First release on PyPI.
- Contains documentation and a stub API for querying, but no working code yet.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

H

I

K

L

M

N

P

R

S

T

V