cf-pandas

Release 0.8.1

Axiom Data Science

Feb 12, 2024

DOCUMENTATION

CHAPTER

ONE

INSTALLATION

To install from conda-forge:

```
>>> conda install -c conda-forge cf_pandas
```

To install from PyPI:

```
>>> pip install cf-pandas
```

1.1 How to use `cf-pandas`

The main use of `cf-pandas` currently is for selecting columns of a DataFrame that represent axes or coordinates of the dataset and for selecting a variable from a `pandas DataFrame` using the accessor and a custom vocabulary that searches column names for a match to the regular expressions, as well as some other capabilities that have been ported over from `cf-xarray`. There are several class and utilities that support this functionality that are used internally but are also helpful for other packages.

```
import cf_pandas as cfp
import pandas as pd
```

1.1.1 Get some data

```
# Some data
url = "https://files.stage.platforms.axds.co/axiom/netcdf_harvest/basis/2013/BE2013_/
↪data.csv.gz"
df = pd.read_csv(url)
df
```

```
                    time  longitude   latitude     z  profile  temperature  \
0      2013-08-07T22:26:00 -168.01784  65.409500   0.0        0      10.3291
1      2013-08-07T22:26:00 -168.01784  65.409500  66.0        0          NaN
2      2013-08-07T22:26:00 -168.01784  65.409500  65.0        0          NaN
3      2013-08-07T22:26:00 -168.01784  65.409500  64.0        0          NaN
4      2013-08-07T22:26:00 -168.01784  65.409500  63.0        0          NaN
...                    ...        ...        ...   ...      ...          ...
12735  2013-09-24T22:59:00 -168.01384  60.516167  25.0      139          NaN
12736  2013-09-24T22:59:00 -168.01384  60.516167  24.0      139          NaN
12737  2013-09-24T22:59:00 -168.01384  60.516167  23.0      139          NaN
```

(continues on next page)

(continued from previous page)

```
12738  2013-09-24T22:59:00 -168.01384  60.516167  32.0      139          NaN
12739  2013-09-24T22:59:00 -168.01384  60.516167  90.0      139          NaN

       pressure  salinity  chlorophyll_a  conductivity     distance  segment
0           0.0   30.7286            NaN           NaN         0.00        0
1           NaN       NaN            NaN           NaN         0.00        0
2           NaN       NaN            NaN           NaN         0.00        0
3           NaN       NaN            NaN           NaN         0.00        0
4           NaN       NaN            NaN           NaN         0.00        0
...         ...       ...            ...           ...          ...      ...
12735       NaN       NaN            NaN           NaN  15575752.91        0
12736       NaN       NaN            NaN           NaN  15575752.91        0
12737       NaN       NaN            NaN           NaN  15575752.91        0
12738       NaN       NaN            NaN           NaN  15575752.91        0
12739       NaN       NaN            NaN           NaN  15575752.91        0

[12740 rows x 12 columns]
```

1.1.2 Basic accessor usage

The terminology all comes from `cf-xarray` which deals with multi-dimensional data and has more layers of standardized attributes. This package ports over useful functionality, retaining some of the complexity of terminology and syntax from `cf-xarray` which doesn't always apply. The perspective is to be able to think about and use DataFrames of data in a similar manner to Datasets of data/model output.

When you use the `cf-pandas` accessor it will first validate that columns representing time, latitude, and longitude are present and identifiable (by validating the object).

Using an approach copied directly from `cf-xarray`, `cf-pandas` contains a mapping of names from the CF conventions that define the axes ("T", "Z", "Y", "X") and coordinates ("time", "vertical", "latitude", "longitude"). These are built in and used to identify columns containing axes and coordinates using name matching (column names are split by white space for the comparison).

Check axes and coordinates mappings of the dataset:

```
df.cf.axes, df.cf.coordinates
```

```
({'Z': ['z'], 'T': ['time']},
 {'longitude': ['longitude'], 'latitude': ['latitude'], 'time': ['time']})
```

Check all available keys:

```
df.cf.keys()
```

```
{'T', 'Z', 'latitude', 'longitude', 'time'}
```

Is a certain key in the DataFrame?

```
"T" in df.cf, "X" in df.cf
```

```
(True, False)
```

What CF standard names can be identified with strict matching in the column names? Column names will be split by white space for this comparison.

```
df.cf.standard_names
```

```
{'latitude': ['latitude'], 'longitude': ['longitude'], 'time': ['time']}
```

1.1.3 Select variable

Selecting a variable typically requires knowing the name of the column representing the variable. What is demonstrated here is an approach to selecting a column name containing the variable using regular expression matching. In this case, the user defines the regular expression matching that will be used to identify matches to a variable. There are helper functions for this process available in `cf-pandas`; see the `Reg`, `Vocab`, and `widget` classes and below for more information.

Create custom vocabulary

More information about custom vocabularies and using the `Vocab` class here: https://cf-pandas.readthedocs.io/en/latest/demo_vocab.html

You can make regular expressions for your vocabulary by hand or use the `Reg` class in `cf-pandas` to do so.

```
# initialize class
vocab = cfp.Vocab()

# define a regular expression to represent your variable
reg = cfp.Reg(include="salinity", exclude="soil", exclude_end="_qc")

# Make an entry to add to your vocabulary
vocab.make_entry("salt", reg.pattern(), attr="standard_name")

# Add another entry to vocab
vocab.make_entry("temp", "temp")

vocab
```

```
{'salt': {'standard_name': '(?i)^(?!.*(soil))(?!.*(_qc)$)(?=.*salinity)'}, 'temp': {
↪'standard_name': 'temp'}}
```

Access variable

Refer to the column of data you want by the nickname described in your custom vocabulary.

You can do this with a context manager, especially if you are using more than one vocabulary:

```
with cfp.set_options(custom_criteria=vocab.vocab):
    print(df.cf["salt"])
```

```
0        30.7286
1            NaN
2            NaN
```

(continues on next page)

(continued from previous page)

```
3            NaN
4            NaN
          ...
12735        NaN
12736        NaN
12737        NaN
12738        NaN
12739        NaN
Name: salinity, Length: 12740, dtype: float64
```

Or you can set one for use generally in this kernel:

```
cfp.set_options(custom_criteria=vocab.vocab)
df.cf["salt"]
```

```
0        30.7286
1            NaN
2            NaN
3            NaN
4            NaN
          ...
12735        NaN
12736        NaN
12737        NaN
12738        NaN
12739        NaN
Name: salinity, Length: 12740, dtype: float64
```

Display mapping of all variables in the dataset that can be identified using the custom criteria/vocab we defined above:

```
df.cf.custom_keys
```

```
{'salt': ['salinity'], 'temp': ['temperature']}
```

1.1.4 Other utilities

Access all CF Standard Names

```
sn = cfp.standard_names()
sn[:5]
```

```
['acoustic_signal_roundtrip_travel_time_in_sea_water',
 'aerodynamic_particle_diameter',
 'aerodynamic_resistance',
 'age_of_sea_ice',
 'age_of_stratospheric_air']
```

Use vocabulary to match any list

This is the logic under the hood of the `cf-pandas` accessor that selects what column matches a variable nickname according to the custom vocabulary. This comes from `cf-xarray` almost exactly. It is available as a separate function because it is useful to use in other scenarios too. Here we filter the standard names just found by our custom vocabulary from above.

```
cfp.match_criteria_key(sn, "salt", vocab.vocab)
```

```
['sea_water_practical_salinity_at_sea_floor',
 'tendency_of_sea_water_salinity',
 'sea_water_absolute_salinity',
 'tendency_of_sea_water_salinity_expressed_as_salt_content',
 'change_over_time_in_sea_water_preformed_salinity',
 'tendency_of_sea_water_salinity_due_to_vertical_mixing',
 'tendency_of_sea_water_salinity_due_to_sea_ice_thermodynamics',
 'sea_water_salinity',
 'tendency_of_sea_water_salinity_expressed_as_salt_content_due_to_parameterized_
↪submesoscale_eddy_advection',
 'square_of_sea_surface_salinity',
 'sea_water_cox_salinity',
 'integral_wrt_depth_of_product_of_salinity_and_sea_water_density',
 'sea_water_practical_salinity',
 'tendency_of_sea_water_salinity_expressed_as_salt_content_due_to_parameterized_eddy_
↪advection',
 'tendency_of_sea_water_salinity_due_to_horizontal_mixing',
 'tendency_of_sea_water_salinity_expressed_as_salt_content_due_to_parameterized_
↪mesoscale_eddy_advection',
 'integral_wrt_depth_of_sea_water_practical_salinity',
 'tendency_of_sea_water_salinity_expressed_as_salt_content_due_to_parameterized_
↪mesoscale_eddy_diffusion',
 'sea_surface_salinity',
 'change_over_time_in_sea_water_absolute_salinity',
 'tendency_of_sea_water_salinity_due_to_parameterized_eddy_advection',
 'ratio_of_sea_water_practical_salinity_anomaly_to_relaxation_timescale',
 'tendency_of_sea_water_salinity_expressed_as_salt_content_due_to_parameterized_
↪dianeutral_mixing',
 'product_of_eastward_sea_water_velocity_and_salinity',
 'product_of_northward_sea_water_velocity_and_salinity',
 'tendency_of_sea_water_salinity_expressed_as_salt_content_due_to_residual_mean_advection
↪',
 'sea_water_salinity_at_sea_floor',
 'tendency_of_sea_water_salinity_due_to_advection',
 'sea_water_reference_salinity',
 'change_over_time_in_sea_water_practical_salinity',
 'sea_water_knudsen_salinity',
 'sea_water_preformed_salinity',
 'change_over_time_in_sea_water_salinity',
 'sea_ice_salinity']
```

1.2 Reg: Write Regular Expressions

This class will help you write simple regular expressions. The available options are:

These all work as logical "or" if there is more than one string specified:

- `exclude` (string or list)
- `exclude_start` (string or list)
- `exclude_end` (string or list)

Also:

- `include` (logical "and", string or list)
- `include_or` (logical "or", string or list)
- `include_exact` (string)
- `include_start` (string)
- `include_end` (string)
- and `ignore_case` (bool)

If you find you want to use more than one `include_exact`, `include_start`, or `include_end` at once, you should write a new regular expression with the class, instead. That is, write multiple expressions and pipe them together with a pipe between, like:

`"expression1|expression2"`

rather than try to get everything into a single expression, or just use the built-in convenience function:

`cfp.joinpat([reg1, reg2])`

Note: you may need to use Python package `regex` instead of `re` with piped-together expressions.

```
import cf_pandas as cfp
import regex
```

1.2.1 Write a regular expression

```
reg = cfp.Reg(include="one", exclude="two")
reg.pattern()
```

```
'(?i)^(?!.*(two))(?=.*one)'
```

```
[string for string in ["onetwo","twothree","onethree"] if regex.match(reg.pattern(),␣
↪string)]
```

```
['onethree']
```

1.3 Vocab: manage custom vocabularies

Custom vocabularies are used in `cf-xarray` and `cf-pandas` to search variable names and available metadata (in the case of `cf-xarray` with Dataset variable attributes) and compare with regular expressions to make selections. These make it possible to generically select variables from different Datasets and DataFrames. However, they are difficult to write, handle, and maintain, and control which variables are found so being able to tweak them is important.

```
import cf_pandas as cfp
import regex
```

1.3.1 What does a custom vocabulary look like?

cf-xarray

For a netcdf-style Dataset, this criteria dictionary will be used to search over the attributes in each variable and compare with the regular expressions to search for matches, in this case checking specifically the variable `standard_name` and `name`. There will be a match to the nickname "ssh" if the `standard_name` for a variable is exactly "sea_surface_height" or if the `standard_name` contains the string "sea_surface_elevation".

cf-pandas

`cf-pandas` is made to access `pandas` DataFrames in an analogous manner to using the `cf-xarray` accessor with Datasets. DataFrames do not have attributes to compare with, only the column name. So, every regular expression for a given nickname (in the example given, "ssh" and "temp"), will be compared with the column name. The extra dictionary structure is maintained in this case so that vocabularies can be used between `cf-xarray` and `cf-pandas` if needed, for example, to select a model variable and compare it with a data file variable.

```
criteria = {
    "ssh": {
        "standard_name": "sea_surface_height$|sea_surface_elevation",
        "name": "(?i)sea_surface_elevation(?!.*?_qc)"
    },
    "temp": {
        "standard_name": "sea_water_temperature",
        "name": "(?i)temperature(?!.*(skin|ground|air|_qc))"
    },
}
criteria
```

```
{'ssh': {'standard_name': 'sea_surface_height$|sea_surface_elevation',
  'name': '(?i)sea_surface_elevation(?!.*?_qc)'},
 'temp': {'standard_name': 'sea_water_temperature',
  'name': '(?i)temperature(?!.*(skin|ground|air|_qc))'}}
```

1.3.2 How should I use the custom vocabulary?

You can set your vocabulary to be used generally, or use with a context-manager. I recommend using the context-manager approach whenever you might use more than one vocabulary.

To set it generally for, for example, `cf-xarray`, you would do:

`cf_xarray.set_options(custom_criteria=criteria)`

or for `cf-pandas`, imported as `cfp`:

`cfp.set_options(custom_criteria=criteria)`

In this demo, we will use the context manager approach so that we can use more than one vocabulary. For example, here we compare a list of strings with the vocabulary we called `criteria` and are searching for all matches in the list to the variable by the nickname "ssh".

```
vals = ["zeta", "sea_surface_height", "sea_surface", "sea_surface_elevation_zeta"]

with cfp.set_options(custom_criteria=criteria):
    print(cfp.match_criteria_key(vals, "ssh"))
```

```
['sea_surface_elevation_zeta', 'sea_surface_height']
```

1.3.3 How can I make a custom vocabulary? Introducing the Vocab class.

As you can see, making `criteria` could be labor intensive. Also, users may want to create more than one of these and use them separately or together in different circumstances, and save them for later. That is what the `Vocab` class in `cf-pandas` is meant to help with.

In this example, we make an entry in our vocabulary that has two regular expressions in it to start.

```
# initialize class
vocab1 = cfp.Vocab()

# Make an entry to add to your vocabulary
vocab1.make_entry("new_variable_nickname", ["match_this_exactly$", "match_that_exactly$
↪"], attr="name")
```

```
{'new_variable_nickname': {'name': 'match_this_exactly$|match_that_exactly$'}}
```

Retrieve the vocabulary you've made with `vocab1.vocab`:

```
vocab1.vocab
```

```
defaultdict(dict,
            {'new_variable_nickname': {'name': 'match_this_exactly$|match_that_exactly$'}
↪})
```

And you can subsequently add other entries:

```
# add another entry
vocab1.make_entry("new_variable_nickname", ["match_this_string", "match_that_exactly$"],␣
↪attr="long_name")
```

(continues on next page)

(continued from previous page)

```
# add another entry
vocab1.make_entry("other_variable_nickname", "match_that_string", attr="standard_name")
```

```
{'new_variable_nickname': {'name': 'match_this_exactly$|match_that_exactly$', 'long_name
↪': 'match_this_string|match_that_exactly$'}, 'other_variable_nickname': {'standard_name
↪': 'match_that_string'}}
```

```
vocab1
```

```
{'new_variable_nickname': {'name': 'match_this_exactly$|match_that_exactly$', 'long_name
↪': 'match_this_string|match_that_exactly$'}, 'other_variable_nickname': {'standard_name
↪': 'match_that_string'}}
```

Test our new vocabulary:

```
vals = ["match_this_exactly", "match_this_exactly_but", "other_variable_nickname",
↪"match_that_string"]

with cfp.set_options(custom_criteria=vocab1.vocab):
    print(cfp.match_criteria_key(vals, "new_variable_nickname"))
```

```
['match_this_exactly']
```

1.3.4 Working with vocabularies

Save to file

Save your new vocabulary to a file with:

```
vocab1.save(filename)
```

Read from file

Retrieve your previously-saved vocabulary by inputting the path into a new instantiation of the Vocab class with:

```
vocab_read = cfp.Vocab(filepath)
```

Combine

```
vocab1 = cfp.Vocab()
vocab1.make_entry("new_variable_nickname", ["match_this_exactly$", "match_that_exactly$
↪"], attr="name")

vocab2 = cfp.Vocab()
vocab2.make_entry("new_variable_nickname", ["match_this_string", "match_that_exactly$"],␣
↪attr="long_name")
vocab2.make_entry("other_variable_nickname", "match_that_string", attr="standard_name")

vocab1 + vocab2
```

```
{'other_variable_nickname': {'standard_name': 'match_that_string'}, 'new_variable_
↪nickname': {'name': 'match_this_exactly$|match_that_exactly$', 'long_name': 'match_
↪this_string|match_that_exactly$'}}
```

Merge 2 or more Vocab objects:

```
cfp.merge([vocab1, vocab2])
```

```
{'other_variable_nickname': {'standard_name': 'match_that_string'}, 'new_variable_
↪nickname': {'name': 'match_this_exactly$|match_that_exactly$', 'long_name': 'match_
↪this_string|match_that_exactly$'}}
```

Can also add in place

```
# also works
vocab1 += vocab2
```

1.3.5 Use the Reg class to write regular expressions

We used simple exact matching regular expressions above, but for anything more complicated it can be hard to write regular expressions. You can use the `Reg` class in `cf-pandas` to write regular expressions with several options, as demonstrated more in another doc page, and briefly here.

```
# initialize class
vocab = cfp.Vocab()

# define a regular expression to represent your variable
reg = cfp.Reg(include="temperature", exclude="air", exclude_end="_qc", include_start="sea
↪")

# Make an entry to add to your vocabulary
vocab.make_entry("temp", reg.pattern(), attr="standard_name")

vocab
```

```
{'temp': {'standard_name': '(?i)^(?!.*(air))(?!.*(_qc)$)^sea.*(?=.*temperature)'}}
```

1.4 Widget to help humans select strings to match

The best way to understand this demo is with a Binder notebook since it includes a widget! Click on the badge to launch the Binder notebook.

One way to deal with vocabularies (see vocab demo) is to create a vocabulary that represents exactly which variables you want to match with for a given server. This way, when you are interacting with catalogs and data from that server you can be sure that your vocabulary will pull out the correct variables. It is essentially a variable mapping in this use case as opposed to a variable matching.

Sometimes the variables we want to search through for making selections could be very long. This widget is meant to help quickly include and exclude strings from the list and then allow for human-centered multi-select with command/control to export a vocabulary.

```
import cf_pandas as cfp
```

1.4.1 Select from list of CF standard_names

You can read in all available standard_names with a utility in `cf-pandas` with:

`cfp.standard_names()`.

```
names = cfp.standard_names()
```

The basic idea is to write in a nickname for the variable you are representing in the top text box, and then select the standard_names that "count" as that variable. One problem is that if you don't include and exclude specific strings, the list of standard_names is too long to look through and select what you want for a given variable nickname.

Here is an example with a few inputs initialized to demonstrate. You can add more strings to exclude by adding them to the text box with a pipe ("|") between strings like `air|change`. You can pipe together terms to include also; the terms are treated as the logical "or" so the options list will show strings that have at least one of the "include" terms.

Once you narrow the options in the dropdown menu enough, you can select the standard_names you want. When you are happy with your selections, click "Press to save". This creates an entry in the class "vocab" of your variable nickname mapping to the attribute "standard_name" exactly matching each of the standard_names selected. Then, you can enter a new variable nickname and repeat the process to create another entry in the vocabulary.

```
w = cfp.Selector(options=names, nickname_in="temp",
                 exclude_in="air", include_in="temperature")

w.button_pressed()
```

```
interactive(children=(Text(value='temp', description='nickname'), Text(value='temperature
↪', description='inclu...
```

```
Button(description='Press to save', style=ButtonStyle())
```

```
Output()
```

The rest of the notebook shows results based on the user not changing anything in the widget so the results can be consistent.

Look at vocabulary

```
w.vocab
```

```
{'temp': {'standard_name': 'brightness_temperature$'}}
```

Save vocabulary for future use

```
w.vocab.save("std_name_demo")
```

Open and check out your vocab with:

```
cfp.Vocab("std_name_demo")
```

```
{'temp': {'standard_name': 'brightness_temperature$'}}
```

1.5 API

1.5.1 Accessor

From cf-xarray.

class cf_pandas.accessor.**CFAccessor**(*pandas_obj*)

Bases: `object`

Dataframe accessor analogous to cf-xarray accessor.

Attributes

axes
Property that returns a dictionary mapping valid Axis standard names for `.cf[]` to variable names.

axes_cols
Property that returns a list of column names from the axes mapping.

coordinates
Property that returns a dictionary mapping valid Coordinate standard names for `.cf[]` to variable names.

coordinates_cols
Property that returns a list of column names from the coordinates mapping.

custom_keys
Returns a dictionary mapping criteria keys to variable names.

standard_names
Returns a dictionary mapping standard_names to variable names, if there is a match.

Methods

keys()	Utility function that returns valid keys for .cf[].

property axes: Dict[str, List[str]]

Property that returns a dictionary mapping valid Axis standard names for `.cf[]` to variable names.

This is useful for checking whether a key is valid for indexing, i.e. that the attributes necessary to allow indexing by that key exist. It will return the Axis names `("X", "Y", "Z", "T")` present in `.columns`.

Returns
Dictionary with keys that can be used with `__getitem__` or as `.cf[key]`. Keys will be the appropriate subset of ("X", "Y", "Z", "T"). Values are lists of variable names that match that particular key.

Return type
dict

property axes_cols: List[str]

Property that returns a list of column names from the axes mapping.

Returns
Variable names that are the column names which represent axes.

Return type
list

property coordinates: Dict[str, List[str]]

Property that returns a dictionary mapping valid Coordinate standard names for `.cf[]` to variable names.

This is useful for checking whether a key is valid for indexing, i.e. that the attributes necessary to allow indexing by that key exist. It will return the Coordinate names (`"latitude"`, `"longitude"`, `"vertical"`, `"time"`) present in `.columns`.

Returns
Dictionary of valid Coordinate names that can be used with `__getitem__` or `.cf[key]`. Keys will be the appropriate subset of (`"latitude"`, `"longitude"`, `"vertical"`, `"time"`). Values are lists of variable names that match that particular key.

Return type
dict

property coordinates_cols: List[str]

Property that returns a list of column names from the coordinates mapping.

Returns
Variable names that are the column names which represent coordinates.

Return type
list

property custom_keys

Returns a dictionary mapping criteria keys to variable names.

Returns
Dictionary mapping criteria keys to variable names.

Return type
dict

Notes

Need to use this with context manager version of providing custom_criteria.

keys() → Set[str]

Utility function that returns valid keys for .cf[].

This is useful for checking whether a key is valid for indexing, i.e. that the attributes necessary to allow indexing by that key exist.

Returns
Set of valid key names that can be used with __getitem__ or .cf[key].

Return type
set

property standard_names

Returns a dictionary mapping standard_names to variable names, if there is a match. Compares with all cf-standard names.

Returns
Dictionary mapping standard_names to variable names.

Return type
dict

Notes

This is not the same as the cf-xarray accessor method of the same name, which searches for variables with standard_name attributes and surfaces those values to map to the variable name.

cf_pandas.accessor.**apply_mapper**(*mappers: Union[Callable[[DataFrame, str], List[str]], Tuple[Callable[[DataFrame, str], List[str]], ...]]*, *obj: DataFrame*, *key: Hashable*, *error: bool = True*, *default: Optional[Any] = None*) → List[Any]

Applies a mapping function; does error handling / returning defaults. Expects the mapper function to raise an error if passed a bad key. It should return a list in all other cases including when there are no results for a good key.

1.5.2 cf-pandas utilities

Utilities for cf-pandas.

cf_pandas.utils.**always_iterable**(*obj: ~typing.Any*, *allowed=(<class 'tuple'>, <class 'list'>, <class 'set'>, <class 'dict'>)*) → Iterable

This is from cf-xarray.

cf_pandas.utils.**astype**(*value*, *type_*)

Return *value* as type *type_*. Particularly made to work correctly for returning string, *PosixPath*, or *Timestamp* as list.

cf_pandas.utils.**match_criteria_key**(*available_values: list*, *keys_to_match: Union[str, list]*, *criteria: Optional[dict] = None*, *split: bool = False*) → list

Use criteria to choose match to key from available available_values.

Parameters

- **available_values** (*list*) – String or list of strings to compare against list of category values. They should be keys in *criteria*.
- **keys_to_match** (*str, list*) – Key(s) from criteria to match with available_values.
- **criteria** (*dict, optional*) – Criteria to use to map from variable to attributes describing the variable. If user has defined custom_criteria, this will be used by default.
- **split** (*bool, optional*) – If split is True, split the available_values by white space before performing matches. This is helpful e.g. when columns headers have the form "standard_name (units)" and you want to match standard_name.

Returns
Values from available_values that match keys_to_match, according to criteria.

Return type
list

Notes

This uses logic from *cf-xarray*.

cf_pandas.utils.**set_up_criteria**(*criteria: Optional[Union[dict, Iterable]] = None*) → ChainMap

Get custom criteria from options.

Parameters
criteria (*dict, optional*) – Criteria to use to map from variable to attributes describing the variable. If user has defined custom_criteria, this will be used by default.

Returns
Criteria

Return type
ChainMap

cf_pandas.utils.**standard_names**()

Returns list of CF standard_names.

Returns
All CF standard_names

Return type
list

1.5.3 Reg class for writing regular expressions

Class for writing regular expressions.

class cf_pandas.reg.**Reg**(*exclude: Optional[Union[List[str], str]] = None*, *exclude_start: Optional[Union[List[str], str]] = None*, *exclude_end: Optional[Union[List[str], str]] = None*, *include: Optional[Union[List[str], str]] = None*, *include_or: Optional[Union[List[str], str]] = None*, *include_exact: Optional[str] = None*, *include_start: Optional[str] = None*, *include_end: Optional[str] = None*, *ignore_case: bool = True*)

Bases: object

Class to write a regular expression.

Notes

- Input strings are never allowed to be empty.
- Need escape characters on any special characters, and then convert to raw, e.g., r"[celsius]" for "[celsius]".
- The *exclude* options are logical "or".
- The *include* option is logical "and", *include_or* is logical "or", and the other *include_* options allow for only one selection. If you want to use more than one *include_start* for example, you should make an additional regular expression.

Methods

check()	Check to make sure selected options are compatible.
exclude(string)	Exclude string from anywhere in matches.
exclude_end(string)	Exclude string from end of matches.
exclude_start(string)	Exclude string from start of matches.
include(string)	String must be present anywhere in matches, logical "and".
include_end(string)	String must be present at the end of matches.
include_exact(string)	String must match exactly.
include_or(string)	String must be present anywhere in matches, logical "or".
include_start(string)	String must be present at the start of matches.
pattern()	Generate regular expression pattern from user rules.

check()

Check to make sure selected options are compatible.

exclude(*string: Union[str, list]*)

Exclude string from anywhere in matches.

Parameters
string (*str, list*) – Matches with regular expression *pattern* will not contain string(s).

Notes

As a list of strings, this acts as a logical "or" for the exclusions.

exclude_end(*string: Union[str, list]*)

Exclude string from end of matches.

Parameters
string (*str, list*) – Matches with regular expression *pattern* will not end with string(s).

Notes

As a list of strings, this acts as a logical "or" for the exclusions.

exclude_start(*string: Union[str, list]*)

Exclude string from start of matches.

Parameters
string (*str, list*) – Matches with regular expression *pattern* will not start with string(s).

Notes

As a list of strings, this acts as a logical "or" for the exclusions.

include(*string: Union[str, list]*)

String must be present anywhere in matches, logical "and".

Parameters
string (*str, list*) – Matches with regular expression *pattern* will contain all string(s).

Notes

A list of strings will be treated as a logical "and".

include_end(*string: str*)

String must be present at the end of matches.

Parameters
string (*str*) – Matches with regular expression *pattern* will end with string.

include_exact(*string: str*)

String must match exactly.

Parameters
string (*str*) – A match with regular expression *pattern* will be exactly string.

include_or(*string: Union[str, list]*)

String must be present anywhere in matches, logical "or".

Parameters
string (*str, list*) – Matches with regular expression *pattern* will contain at lease one of string(s).

Notes

A list of strings will be treated as a logical "or".

include_start(*string: str*)

String must be present at the start of matches.

Parameters
string (*str*) – Matches with regular expression *pattern* will start with string.

pattern() → str

Generate regular expression pattern from user rules.

Returns
Regular expression accounting for all input selections.

Return type
str

cf_pandas.reg.**joinpat**(*regs: Sequence[Reg]*) → str

Join patterns from Reg objects.

Parameters
regs (*Sequence*) – Reg objects from which *.pattern()* will be used.

Returns
Regular expression patterns from regs joined together with "|"

Return type
str

1.5.4 Vocab class for handling custom variable-selection vocabularies

Class for creating and working with vocabularies.

class cf_pandas.vocab.**Vocab**(*openname: Optional[str] = None*)

Bases: object

Class to handle vocabularies.

Methods

add(other_vocab, method)	Add two Vocab objects together...
make_entry(nickname, expressions[, attr])	Make an entry for vocab.
open_file(openname)	Open previously-saved vocab.
save(savename)	Save to file.

add(*other_vocab: Union[DefaultDict[str, Dict[str, str]],* Vocab*], method: str*) → *Vocab*

Add two Vocab objects together…

by adding their *.vocab`s together. Expressions are piped together but otherwise not changed. This is used for both `__add__* and *__iadd__*.

Parameters

- **other_vocab** (Vocab) – Other Vocab object to combine with.
- **method** (*str*) – Whether to run as "add" which returns a new Vocab object or "iadd" which adds to the original object.

Returns
vocab + other_vocab either as a new object or in place.

Return type
Vocab

make_entry(*nickname: str*, *expressions: Union[str, list]*, *attr: str = 'standard_name'*)

Make an entry for vocab.

Parameters

- **nickname** (*str*) – The nickname to call the variable being represented in this entry.
- **expressions** (*str, list*) – Regular expression(s) to use to select out the variable in a regex match. Multiple expressions input in a list are piped together to create one str of expressions.
- **attr** (*str*) – What attribute to identify the regular expressions with. Default is "standard_name", but other reasonable options are any variable attributes in a netcdf file such as "units", "name", and "long_name".

Examples

The following creates an entry in the vocabulary stored in *vocab.vocab*. It doesn't print the entry but it has been pasted in below the example to show what it looks like.

```
>>> import cf_pandas as cfp
>>> vocab = cfp.Vocab()
>>> vocab.make_entry("temp", ["a","b"], attr="name")
{'temp': {'standard_name': 'a|b'}})
```

open_file(*openname: Union[str, PurePath]*)

Open previously-saved vocab.

Parameters
openname (*str*) – Where to find vocab to open.

save(*savename: Union[str, PurePath]*)

Save to file.

Parameters
savename (*str, PurePath*) – Filename to save to.

cf_pandas.vocab.**merge**(*vocabs: Sequence[*Vocab*]*) → *Vocab*

Add together multiple Vocab objects.

Parameters
vocabs (*Sequence[*Vocab*]*) – Sequence of Vocab objects to merge.

Returns
Single Vocab object made up of input vocabs.

Return type
Vocab

1.5.5 widget class for easy human selection of variables to exactly match

Widget

class cf_pandas.widget.**Selector**(*options: Sequence*, *vocab: Optional[*Vocab*] = None*, *nickname_in: str = ''*, *include_in: str = ''*, *exclude_in: str = ''*)

Bases: object

Coordinates interaction with dropdown widget to make simple vocabularies.

Options are filtered by a regular expression written to reflect the include and exclude inputs, and these are updated when changed and shown in the dropdown. The user should select using *command* or *control* to make multiple options. Then push the "save" button when the nickname and selected options from the dropdown menu are the variables you want to include exactly in a future regular expression search.

Examples

Show widget with a short list of options. Input a nickname and press button to save an entry to the running vocabulary in the object:

```
>>> import cf_pandas as cpf
>>> sel = cfp.Selector(options=["var1", "var2", "var3"])
>>> sel
```

See resulting vocabulary with:

```
>>> sel.vocab
```

Methods

button_pressed(*args)	Saves a new entry in the catalog when button is pressed.

button_pressed(**args*)

Saves a new entry in the catalog when button is pressed.

cf_pandas.widget.**dropdown**(*nickname: str*, *options: Union[Sequence, Series]*, *include: str = ''*, *exclude: str = ''*)

Makes widget that is used by class.

Options are filtered by a regular expression written to reflect the include and exclude inputs, and these are updated when changed and shown in the dropdown. The user should select using *command* or *control* to make multiple options. Then push the "save" button when the nickname and selected options from the dropdown menu are the variables you want to include exactly in a future regular expression search.

Parameters

- **nickname** (*str*) – nickname to associate with the Vocab class vocabulary entry from this, e.g., "temp". Inputting this to the function creates a text box for the user to enter it into.
- **options** (*Sequence*) – strings to select from in the dropdown widget. Will be filtered by include and exclude inputs.
- **include** (*str*) – include must be in options values for them to show in the dropdown. Will update as more are input. To input more than one, join separate strings with "|". For example, to search on both "temperature" and "sea_water", input "temperature|sea_water".
- **exclude** (*str*) – exclude must not be in options values for them to show in the dropdown. Will update as more are input. To input more than one, join separate strings with "|". For example, to exclude both "temperature" and "sea_water", input "temperature|sea_water".

PYTHON MODULE INDEX

C

INDEX

V