pyRACF

Version 0.9.0

Table of Contents

Contents:

pyRACF user documentation

pyRACF background

pyRACF was written for the sole purpose of being able to work with RACF configurations from a python runtime.

At its core pyRACF converts every recordtype from an IRRDBU00-unload into its own Panda DataFrame. These DataFrames can be saved as 'pickle-files' in order to work with the same data without having to parse the IRRDBU00-unload again.

Using the python interactive shells, pyRACF allows you to make queries on your RACF database and explore its structure.

For instance, there's a function called `.revoked()` that will instantly return a dataframe of user records (recordtype 200, USBD) containing all the users that have been revoked.

Release notes

Summary of changes

0.9.1 auto_pickle, GID, UID without leading zeros, UACC and APPLID in CERT and KEYRING

- r=RACF(…)auto_pickle=True) compares timestamp of pickles with unload file uses pickles when newer, or reads unload and updates pickles

- ProfilePublisher methods for GPOMVS, USOMVS, CERT and KEYRING tables

- r.table() now uses the profile publisher method to return a df, instead of the df itself.

- new property _doc_stubs in ProfilePublisher with names of methods that are only there for ducmentation purposes, and should not be called

- .find( ) and .skip( ) accept list specification of values, these must be literals, e.g., r.datasetAccess.find(None,None,['SYS1','SYS2']) r.users.find(OWNER=['SYS1','SYS2'])

0.9.0 (rules, sphinx-autodoc, find, skip)

- rule-based verification of profile fields

  ◦ yaml based specification of rules and domains, normal python dicts are also possible

  ◦ test if values in access list are valid user ID or group names

  ◦ test if values are (in a list of) expected values

  ◦ e.g., all SYS1 profiles must have UACC=NONE, OWNER=SYS1, NONOTIFY

  ◦ e.g. STARTED profiles should specify existing STUSER and STGROUP, or =MEMBER

- find() to select rows from Frame, skip() to exclude rows from result

  ◦ may be used on (chained from) profile, acl and rule Frames

  ◦ supports position column values, and keywords with column names

  ◦ e.g. r.datasets.find('SYS1.**',UACC=['CONTROL','ALTER'])

- ◦ replacing gfilter() and rfilter()

- match() method to select profile matching a dataset name, or resource class and name

  - ◦ may be used on the .datasets, .datasetAccess and .datasetConditionalAccess, and the 3 generals Frames

  - ◦ e.g. r.generals.find('FACILITY').match('BPX.SUPERUSER').acl()

  - ◦ accepts individual names, or a list of names

  - ◦ also supported as keyword parameter in find() and skip()

- stripPrefix() method removes prefix from column names, for easier programming

- r.table(name) returns ProfileFrame for given name, e.g. r.table('DSACC')

- all record types from unload are now converted into ProfileFrames

- xlswriter() converted to pandas pivot table

- alternative for orphans using rules script

- StoopidException converted to TypeError

- deprecated properties (.generic, .genericAccess, .userDistributedMappings) removed

- information from README.md and Wiki moved into docs directory - formatted with sphinx, viewable in github

0.8.7 (grouptree, speed-up)

- grouptree and ownertree are now properties, no longer callables

- accept '*' as a literal value in gfilter( )

- r.connect('SYS1') and r.connect(None,'IBMUSER') return one level index

- less contentious column name ALL_USER_ACCESS replaces EFFECTIVE_UACC

- speed up single profile methods

- Single value selections return dataframe with columns again

- giveMeProfiles, generic2regex are now 'internal' (_) functions

0.8.5 (fixes for pickles, pytest, wiki)

- parse_fancycli now creates empty data frames for pickles it could not find

- index added to data frames from old pickles, not for pickles that already have index fields

- pytest framework for QA in development cycle, initial tests ensure attributes are the same with all 3 methods to obtain RACF profiles

- wiki https://github.com/wizardofzos/pyracf/wiki

0.8.3 (tree print format for grouptree and ownertree)

- msys.grouptree and msys.ownertree are now properties instead of callables

  - print as unix tree or simple format, e.g. print(msys.ownertree)

  - default format looks like unix tree, change with msys.ownertree.setformat('simple')

  - dict structure accessible through .tree attribute

- .connect('group') and .connect(None,'user') return a (single level) Index with user IDs, resp., groups, connected to the given entity

  this helps with queries that test group membership

- add IDSTAR_ACCESS and ALL_USER_ACCESS to .datasets and .generals with, resp., permitted access on ID(*) and the higher value of UACC and IDSTAR_ACCESS.

- fixed: correlate also builds internal tables from saved pickles

0.8.2 (property for most application segments)

- application segments for dataset, group and user entities are avaible with entity prefix, e.g., msys.userTSO, msys.datasetDFP, msys.groupOMVS

- system related application segments from general resources are published without prefix, e.g., msys.STDATA or msys.MFA

- old properties msys.installdata and msys.userDistributedMappings are replaced by userUSRDATA and userDistributedIdMappings

- most of these properties are automatically generated

0.8.0 (acl, gfilter, rfilter methods)

- selection method gfilter (supports RACF generic patterns) and rfilter (for regex patterns)

  supports as many parameters as there are index columns in the frame

- reporting method acl, produces frame with access list, may be used on the entity frames or on the access frames

- internal frames _connectByGroup and _connectByUser, as alternate index on connectData

- internal frames _grouptreeLines and _ownertreeLines that return all groups up until SYS1 (or upto a user ID)

- correlate() invoked automatically by parse() and fancycli()

0.7.1 (General instead of Generic)

- fixed: Generic should be General resource profiles, variables and methods renamed

- Mapping between IRRDBU00 record types and variables centralized in a dict

- global constants related to record types removed

- parsed records internally stored by (decimal) record type, instead of by name

- add method to retrieve parsed record count for given name

- _offsets now a RACF class attribute

- fixed: pickles with a prefix were selected when loading pickles with no prefix

- fixed: status property crashed when used before parse() method used, math.floor call is now conditional

- fixed: record type '0260' in offset.json was malformed

- updated offsets.json from IBM documentation

- getOffsets.py to update the json model

- fixed: RACF documentation contains incorrect record type 05k0

- all known record types parsed and loaded into DataFrames

- index columns assigned to all DataFrames, assigned by new correlate() method

- new method correlate() to increase speed of subsequent data access, used after parse() or loading of pickles

- new selection methods similar to user() and group(), that work on index fields.

  - when a parameter is given as None or '**', elements matching the other parameters are returned:

    - datasetPermit and datasetConditionalPermit, with parameters profile(), id() and access()

    - generalPermit and generalConditionalPermit, with parameters resclass(), profile(), id() and access()

    - connect with parameters group() and user()

- added GPMEM_AUTH to connectData frame, consolidating all connect info into one line

0.6.4 (Add 0209)

- Added 0209 recordtype to parser. (userDistributedMapping)

0.6.3 (Add fields)

- Added missing USBD_LEG_PWDHIST_CT, USBD_XPW_PWDHIST_CT, USBD_PHR_ALG, USBD_LEG_PHRHIST_CT, USBD_XPW_PHRHIST_CT, USBD_ROAUDIT and USBD_MFA_FALLBACK to Users dataframe

0.6.2 (Fix XLSX Creation)

- With newer versions of XlsxWriter there's no more .save(). Changed to .close()

- Pinned pandas and XlsxWriter versions in setup.py

0.6.1 (Bug free?)

- XLS generation fully functional again (also for z/VM unloads)

- Oprhan detection working again

- Conditional Dataset Access Records now parsing correctly

- Conditional Dataset Access now correctly pickled :)

- Fixed parsing of GRCACC records (had misparsed AUTH_ID)

- Conditional Generic (General) Records now with correct column name (GRCACC_CLASS_NAME)

0.5.4 (Even more recordtypes!!)

- new property: genericConditionalAccess. Will show GRCACC records.

- Fixed some nasty 'default recordtypes' bugs

0.5.0 (Pickle FTW!)

- new function: save_pickles(path=path, prefix=prefix). Will save all parsed dataframes as pickles (/path/_prefix_*RECORDTYPE*.pickle)

- Can now initialize RACF object from pickle-folder/prefix. To reuse earlier saves pickle files. See examples below

- parse_fancycli now has two optional arguments (save_pickles and prefix) to also save pickle files after parsing to the directory as specified in save_pickles. The prefix argument is only useed with save_pickles isn't False

0.4.5 (Fix Community Update Bug, thanks @Martydog)

- Group Connections now actually usable :)

0.4.4

- Internal constants for all recordtypes

- Improved 'parse_fancycli()'

0.4.3 (Community Update, thanks @Martydog)

- Add User Group Connections record 203

- Add User Installation Data record 204

0.4.2

- Now XLS generation has more checks (fails gracefully if not all required records parsed, works when only genericAccess parsed)

- Same for Orphan detection

- Recordtype 0503 (General Resource Members/genericMembers) added

0.4.2

Installation

Because pyRACF is hosted at pypi.org you can install it easily via:

```
pip install pyracf
```

If you'd rather install from source you need to clone this repository then run the setup.py as follows:

```
git clone https://github.com/wizardofzos/pyracf.git
cd pyracf
python setup.py install
```

Full control installation

If you did not install git on your system, or you need full control over the location of the install directories for pyRACF, you can do the following.

1. Go to the github repository , click the green `< > code` push button, and select `Download ZIP` .

2. Open the ZIP file, it contains one directory `pyracf-main` . Extract this directory, for example, to your Documents folder. When you explore this folder, the executable files can be found in `Documents/pyracf-main/src/pyracf` . This `pyracf` directory is referred to as the `pyracf module` .

3. Add the new directory to your PYTHONPATH, before starting python:

```
export PYTHONPATH=/home/your-id/Documents/pyracf-main/src/:
$PYTHONPATH
```

If you don't have a command prompt because, for example, your start Jupyter Notebook from the desktop, you can add the path containing the pyracf module to the python path with python commands:

```
import sys
new_path = '/home/your-id/Documents/pyracf-main/src/'
if new_path not in sys.path:
    sys.path.insert(0,new_path)
```

4. If this worked, you should be able to load the main class with:

```python
from pyracf import RACF
```

Now, this may fail with an `ModuleNotFoundError` error, when the pre-requisites have not been installed. At the end of the traceback you might see an error message similar to:

```
ModuleNotFoundError: No module named 'xlsxwriter'
```

For each of the modules listed, in this case `xlsxwriter` , issue a pip3 command in the command window:

```
pip3 install xlsxwriter
```

To inspect the list of libraries python uses, type:

```python
import sys
sys.path
```

Using this mechanism, you could also switch between different versions of the pyracf module easily, without having to learn about `venv` . Just remember to restart your python kernel, so you get a new path and fresh modules.

If you're stuck in the python command prompt, you can exit by typing *exit()* .

Jupyter Notebook

The python (or python3) command presents you with a command prompt where you enter a python command and press Enter to see the result. The cursor-up key retrieves previous commands that you can modify before running them again. These commands are saved across sessions too. You can also scroll up and down through the command output using the scroll bar at the right edge of your screen. However, the command prompt is not really an editor.

Jupyter Notebook allows you to enter commands into *cells* , you execute a cell by pressing shift-Enter You leave the code in the previous cells just where it is, and enter extra lines in the next empty cell. And in fact, you can review output from those previous cells without having to re-run them.

Notebooks can be saved, look for the floppy disk icon. You can archive a notebook, by saving it under another name, you can duplicate notebooks from the notebook directory page. You might

say, notebook are to the python command prompt what a desktop environment is to the DOS command prompt.

Read more here or here .

If you wanted a more complete IDE, you can also install JupyterLab using `pip3 install jupyterlab` .

Load RACF data

Unload a RACF database

In order to work with RACF-unloads you'll first need an IRRDBU00-unload. You can create this with the following JCL

```
//UNLOAD    EXEC PGM=IRRDBU00,PARM=NOLOCKINPUT
//SYSPRINT   DD SYSOUT=*
//INDD1    DD   DISP=SHR,DSN=PATH.TO.YOUR.RACFDB
//OUTDD    DD   DISP=(,CATLG,DELETE),
//              DSN=SYS1.RACFDB.FLATFILE,
//              DCB=(RECFM=VB,LRECL=4096),
//              SPACE=(CYL,(50,150),RLSE)
```

For more information on IRRDBU00 head on over to the IBM Documentation .

Once you have your RACF database in IRRDBU00 format, send this over to the workstation where pyRACF is installed.

Parsing

```
>>> from pyracf import RACF
>>> r = RACF(irrdbu00='/path/to/irrdbu00')
```

At this point in time, pyRACF is aware of your unload file, you can check this via

```
>>> r.status
{'status': 'Initial Object', 'input-lines': 7137540, 'lines-read':
0, 'lines-parsed': 0, 'lines-per-second': 'n.a.', 'parse-time':
'n.a.'}
```

Once you tell pyRACF to start parsing your unload (via the `.parse()` function) you can check the status via the `.status` property again and again, until it's done.

```
>>> r.parse()
>>> r.status
```

```
{'status': 'Still parsing your unload', 'input-lines': 7137540,
'lines-read': 894700, 'lines-parsed': 894696, 'lines-per-second':
599275, 'parse-time': 'n.a.'}
>>> r.status
{'status': 'Ready', 'input-lines': 7137540, 'lines-read': 7137540,
'lines-parsed': 7137533, 'lines-per-second': 205447, 'parse-time':
34.741466}
```

As you can see above, the status will be `Ready` once all the records have been parsed and the Panda DataFrames have been built.

A typical parse-block therefore mostly looks something like this:

```
>>> from pyracf import RACF
>>> import time
>>> r = RACF(irrdbu00='/path/to/irrdbu00')
>>> r.parse()
>>> while r.status['status'] != "Ready":
...     print('Parsing...')
...     time.sleep(10)
...
Parsing...
Parsing...
>>> r.status()
{'status': 'Ready', 'input-lines': 7137540, 'lines-read': 7137540,
'lines-parsed': 7137533, 'lines-per-second': 211951, 'parse-time':
33.6753}
```

Fancy Parsing

For a command-line interface the pyRACF module has a `.parse_fancycli()` method that will implement the above loop in a somewhat graphical manner. It will show you all the record types that are selected for parsing and give you a nice overview at the end.

```
>>> r = RACF(irrdbu00='/path/to/irrdbu00')
>>> r.parse_fancycli()
24-04-07 16:53:28 - parsing pyracf/irrdbu00
24-04-07 16:53:28 - selected recordtypes: 0100,0101,...
24-04-07 16:53:55 - progress:
▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊
(100.00%)
24-04-07 16:54:07 - recordtype 0100 -> 23991 records parsed
24-04-07 16:54:07 - recordtype 0101 -> 23990 records parsed
...
24-04-07 16:54:07 - total parse time: 38.597577 seconds
```

Pickles

After parsing the input from the unload file, the RACF object has up to 100 pandas DataFrames. For future runs, you can save these DataFrames as `pickle files` . This should save processing time when you have to build another RACF object:

```
>>> r = RACF(irrdbu00='/path/to/irrdbu00')
>>> r.parse_fancycli(save_pickles='/tmp', prefix='prodsys-')
```

This creates up to 100 files in /tmp, with names like prodsys-GPBD.pickle and prodsys-USOMVS.pickle. Although `prefix` is optional, it helps to identify the files in future.

If you use the parse() method, you can execute a save_pickles() method too:

```
>>> r = RACF(irrdbu00='/path/to/irrdbu00')
>>> r.parse()
>>> while r.status['status'] != "Ready":
...    print('Parsing...')
...    time.sleep(10)
...
Parsing...
Parsing...
>>> r.save_pickles(path='/tmp', prefix='prodsys-')
```

Next time, when you start pyRACF, instead of reading the unload file, you can simply load the pickles, like so:

```
>>> r = RACF(pickles='/tmp', prefix='prodsys-')
```

When you find that pickles are a time-saver, but you forget to refresh the pickles every time a new unload file is created, you can use the `auto-pickles` function:

```
>>> r = RACF(irrdbu00='/path/to/irrdbu00', pickles='/tmp',
prefix='prodsys-', auto_pickle=True)
```

This compares the timestamp of the pickles in the indicated directory with the timestamp of the unload file. If the unload is newer, or there are no matching pickles, the unload is read and the pickles are written. Next time, when the unload is older than the pickles, they are read. Automatically.

Data Table Properties

pyRACF dynamically creates a property for every recordtype it parses from the IRRDBU00 unload. The properties return a DataFrame of the recordtype with column names the same as those in the IBM Documentation . For instance, the unloaded basic-user-information (record Type 200) will have a column name USBD_NAME to contain the "User ID as taken from the profile name".

The following properties directly relate to the recordtypes, and mostly have field names starting with the value under Prefix.

Record types and properties

Type	Prefix	Property	Description
0100	GPBD	groups	Group Basic Data
0101	GPSGRP	subgroups	Group Subgroups
0102	GPMEM	connects	Group Members
0103	GPINSTD	groupUSRDATA	Group Installation Data
0110	GPDFP	groupDFP	Group DFP Data
0120	GPOMVS	groupOMVS	Group OMVS Data
0130	GPOVM	groupOVM	Group OVM Data
0141	GPTME	groupTME	Group TME Data
0151	GPCSD	groupCSDATA	Group CSDATA custom fields
0200	USBD	users	User Basic Data

Type	Prefix	Property	Description
0201	USCAT	userCategories	User Categories
0202	USCLA	userClasses	User Classes
0203	USGCON	groupConnect	User Group Connections
0204	USINSTD	userUSRDATA	User Installation Data
0205	USCON	connectData	User Connect Data
0206	USRSF	userRRSFDATA	RRSF data
0207	USCERT	userCERTname	user certificate name
0208	USNMAP	userAssociationMapping	User Associated Mappings
0209	USDMAP	userDistributedIdMapping	User Associated Distributed Mappings
020A	USMFA	userMFAfactor	user Multifactor authentication data
020B	USMPOL	userMFApolicies	user Multi-factor authentication policies
0210	USDFP	userDFP	User DFP data
0220	USTSO	userTSO	User TSO Data
0230	USCICS	userCICS	User CICS Data

Type	Prefix	Property	Description
0231	USCOPC	userCICSoperatorClasses	User CICS Operator Class
0232	USCRSL	userCICSrslKeys	User CICS RSL keys
0233	USCTSL	userCICStslKeys	User CICS TSL keys
0240	USLAN	userLANGUAGE	User Language Data
0250	USOPR	userOPERPARM	User OPERPARM Data
0251	USOPRP	userOPERPARMscope	User OPERPARM Scope
0260	USWRK	userWORKATTR	User WORKATTR Data
0270	USOMVS	userOMVS	User Data
0280	USNETV	userNETVIEW	user NETVIEW segment
0281	USNOPC	userNETVIEWopclass	user OPCLASS
0282	USNDOM	userNETVIEWdomains	user DOMAINS
0290	USDCE	userDCE	user DCE data
02A0	USOVM	userOVM	user OVM data
02B0	USLNOT	userLNOTES	LNOTES data

Type	Prefix	Property	Description
02C0	USNDS	userNDS	NDS data
02D0	USKERB	userKERB	User KERB segment
02E0	USPROXY	userPROXY	user PROXY
02F0	USEIM	userEIM	user EIM segment
02G1	USCSD	userCSDATA	user CSDATA custom fields
1210	USMFAC	userMFAfactorTags	user Multifactor authentication factor configuration data
0400	DSBD	datasets	Data Set Basic Data
0401	DSCAT	datasetCategories	Data Set Categories
0402	DSCACC	datasetConditionalAccess	Data Set Conditional Access
0403	DSVOL	datasetVolumes	Data Set Volumes
0404	DSACC	datasetAccess	Data Set Access
0405	DSINSTD	datasetUSRDATA	Data Set Installation Data
0406	DSMEM	datasetMember	Data Set Member Data
0410	DSDFP	datasetDFP	Data Set DFP Data

Type	Prefix	Property	Description
0421	DSTME	datasetTME	Data Set TME Data
0431	DSCSD	datasetCSDATA	Data Set CSDATA custom fields
0500	GRBD	generals	General Resource Basic Data
0501	GRTVOL	generalTAPEvolume	General Resource Tape Volume Data
0502	GRCAT	generalCategories	General Resources Categories
0503	GRMEM	generalMembers	General Resource Members
0504	GRVOL	generalTAPEvolumes	General Resources Volumes
0505	GRACC	generalAccess	General Resource Access
0506	GRINSTD	generalUSRDATA	General Resource Installation Data
0507	GRCACC	generalConditionalAccess	General Resources Conditional Access
0508	GRFLTR	DistributedIdFilter	Filter Data
0509	GRDMAP	DistributedIdMapping	General Resource Distributed Identity Mapping Data
0510	GRSES	SESSION	General Resources Session Data
0511	GRSESE	SESSIONentities	General Resources Session Entities

Type	Prefix	Property	Description
0520	GRDLF	DLFDATA	General Resources DLF Data
0521	GRDLFJ	DLFDATAjobnames	General Resources DLF Job Names
0530	GRSIGN	SSIGNON	SSIGNON data
0540	GRST	STDATA	STARTED Class
0550	GRSV	SVFMR	Systemview
0560	GRCERT	CERT	Certificate Data
1560	CERTN	CERTname	general resource certificate information
0561	CERTR	CERTreferences	Certificate References
0562	KEYR	KEYRING	Key Ring Data
0570	GRTME	TME	general resource TME data
0571	GRTMEC	TMEchild	general resource TME child
0572	GRTMER	TMEresource	general resource TME resource
0573	GRTMEG	TMEgroup	general resource TME group
0574	GRTMEE	TMErole	general resource TME role
0580	GRKERB	KERB	general resource KERB segment

Type	Prefix	Property	Description
0590	GRPROXY	PROXY	general resource PROXY
05A0	GREIM	EIM	general resource EIM segment
05B0	GRALIAS	ALIAS	general resource ALIAS group
05C0	GRCDT	CDTINFO	general resource CDTINFO data
05D0	GRICTX	ICTX	general resource ICTX segment
05E0	GRCFDEF	CFDEF	general resource CFDEF data
05F0	GRSIG	SIGVER	general resource SIGVER data
05G0	GRCSF	ICSF	general resource ICSF
05G1	GRCSFK	ICSFsymexportKeylabel	general resource ICSF key label
05G2	GRCSFC	ICSFsymexportCertificateIdentifier	general resource ICSF certificate identifier
05H0	GRMFA	MFA	Multifactor factor definition data
05I0	GRMFP	MFPOLICY	Multifactor Policy Definition data
05I1	GRMPF	MFPOLICYfactors	user Multifactor authentication policy factors
05J1	GRCSD	generalCSDATA	General Resources CSDA custom fields

Type	Prefix	Property	Description
05K0	GRIDTP	IDTFPARMS	Identity Token data
05L0	GRJES	JES	JES data

Properties starting with .general are mostly related to access control profiles that use PERMITs. General resource profiles that represent (system) tables and switches are stored in properties with names that reflect the application segment name (in uppercase, optionally followed by a suffix for lists stored in the segment).

Connect information

Connect information is stored in 3 structures in the RACF database. These structures are represented in 3 properties:

.connects and *.groupConnect* present limited information, *.connects* ignores universal groups, and both lack information about group privileges.

Complete information about connections between groups and users, including connect authority, is stored in *.connectData* .

Extra fields added

Some of these properties have been extended for easier reporting:

.connectData

Combines fields from USER profiles (0205) and GROUP profiles (0102). The `GPMEM_AUTH` field shows group connect authority, whereas all other field names start with `USCON` . This property should be used for most connect group analysis, instead of `.connects` and `.groupConnect` .

.datasets and .generals

Column `IDSTAR_ACCESS` is added by selecting records from `.datasetAccess` and `.generalAccess` referencing ID(*). The higher value of *prefix* _UACC and IDSTAR_ACCESS is stored in

`ALL_USER_ACCESS` indicating the access level granted to all RACF defined users, except when restricted by specific access.

.groupOMVS and .userOMVS

Column `GPOMVS_GID` and `USOMVS_UID` contain the id of the entity in 10 digits with leading zeros, making for errors in specifying a value. A copy of the GID or UID without leading zeros is available in `GPOMVS_GID_` and `USOMVS_UID_` , resp.

.CERT

Returns a combined DataFrame of the DataFrames `._generalCERT` en `.generals` , copying the `GRBD_APPL_DATA` and `GRBD_UACC` fields to show the user ID associated with the certificate and the trust level.

.KEYRING

Returns a combined DataFrame of the DataFrames `._generalKEYRING` en `.generals` , copying the `GRBD_APPL_DATA` field to show the user ID associated with the keyring.

.SSIGNON

Returns a combined DataFrame of the DataFrames `._generalSSIGNON` en `.generals` , copying the `GRBD_APPL_DATA` field to show if replay protection is available for the passticket.

What are the field names?

To view column names in a DataFrame, use `.columns`

```
>>> r.STDATA.columns
Index(['GRST_RECORD_TYPE', 'GRST_NAME', 'GRST_CLASS_NAME',
       'GRST_USER_ID', 'GRST_GROUP_ID', 'GRST_TRUSTED',
       'GRST_PRIVILEGED', 'GRST_TRACE'],
      dtype='object')
```

Data Table Indices

The data tables have index fields assigned to speed up access to entries and to determine "is this ID present in the .users table". Index fields are automatically assigned (generally) as follows. Note that the table prefix is omitted from the index names to ease table processing.

- For tables about groups, users and data sets, the `_NAME` field refers to the profile key.

- For general resources, `_CLASS_NAME` and `_NAME` refer to the resource class and the profile key, resp.

- `.connectData` uses `_GRP_ID` and `_NAME` as index fields, representing the group name and the user ID, resp. The other two connect related tables use the same structure to facilitate merging of tables.

- `.datasetAccess` and `.datasetConditionalAccess` use `_NAME` , `_AUTH_ID` and `_ACCESS` as index fields.

- `.generalAccess` and `.generalConditionalAccess` use `_CLASS_NAME` , `_NAME` , `_AUTH_ID` and `_ACCESS` as index fields.

Tables and views derived from these main tables mostly inherit the index fields. To check the index names used in a DataFrame, use `.index.names`

```
>>> r.STDATA.index.names
FrozenList(['_CLASS_NAME', '_NAME'])
```

Data selection methods

The data table properties from the first section return all profiles and profile data loaded from the RACF input source. Since they typically return more than one entry, the property name represents a plural, such as `.users` . There are 2 options to make selections:

- use standard pandas methods such as `.loc[ ]` and `.query( )` , see Pandas Methods , or

- use RACF specific methods such as `.find( )` , `.skip( )` , `.match( )` , or their deprecated versions `gfilter( )` , and `rfilter( )` , see Selection Methods for guidance and examples.

There is also a range of methods that select one entry from a specific DataFrame, when you know the name of the entry exactly, see Methods to select specific entries .

Analytic Properties

These properties present a subset of a DataFrame, or the result of DataFrame intersections, to identify points of interest.

.specials

The `.specials` property returns a "USBD" DataFrame (like `.users`) with all users that have the 'special attribute' set. Effectively this is the same as the result from

```
r.users.loc[r.users['USBD_SPECIAL'] == 'YES']
```

.operations

Like the `.specials` property but now all the users that have the 'operations attribute' set are returned.

.auditors

Returns a DataFrame with all users that have the 'auditor attribute'

.revoked

Returns a DataFrame with all revoked users.

.groupsWithoutUsers

Returns a DataFrame with all groups that have no user IDs connected (empty groups).

.uacc_read_datasets

Returns a DataFrame with all dataset definitions that have a Universal Access of 'READ'

.uacc_update_datasets

Returns a DataFrame with all dataset definitions that have a Universal Access of 'UPDATE'

.uacc_control_datasets

Returns a DataFrame with all dataset definitions that have a Universal Access of 'CONTROL'

.uacc_alter_datasets

Returns a DataFrame with all dataset definitions that have a Universal Access of 'ALTER'

.orphans

Returns a tuple of `.datasetAccess` DataFrame and `.generalAccess` DataFrame with entries that refer to non-existing authid's.

Methods to select specific entries

In addition to the data table properties, data selection methods are available to retrieve one profile, or profiles from one class, with an easy syntax. The parameter(s) to these methods are used as a literal search argument, and return entries that fully match the argument(s), that means, the selection criteria have to be match the profile exactly.

class pyracf.profile_publishers. ProfileSelectionFrame

    Data selection methods to retrieve one profile, or profiles, from a ProfileFrame, using exact match.

    These methods typically have a name referring to the singular.

    The parameter(s) to these methods are used as a literal search argument, and return entries that fully match the argument(s). Selection criteria have to match the profile exactly, generic patterns are taken as literals.

    The number of selection parameters depends on the ProfileFrame, matching the number of index fields in the ProfileFrame. When you specify a parameter as None or '**', the level is ignored in the selection.

    The optional parameter `option='LIST'` causes a pandas Series to be returned if there is one matching profile, instead of a ProfileFrame. This is meant for high-performance, looping applications.

    group (*group = None* , *option = None*) → ProfileFrame

        data frame with 1 record from `.groups` when the group is found, or an empty frame.

        Example

```
r.group('SYS1')
```

    user (*userid = None* , *option = None*) → ProfileFrame

        data frame with 1 record from `.users` when the user ID is found, or an empty frame.

        Example

```
r.user('IBMUSER')
```

    connect (*group = None* , *userid = None* , *option = None*) → ProfileFrame

        data frame with record(s) from `.connectData` , fitting the parameters exactly, or an empty frame.

        Example

```
r.connect('SYS1','IBMUSER')
```

If one of the parameters is written as `None` , or the second parameter is omitted, all profiles matching the specified parameter are shown, with one index level instead of the 2 index levels that .connectData holds. For example, `r.connect('SYS1')` shows all users connected to SYS1, whereas `r.connect(None, 'IBMUSER')` shows all the groups IBMUSER is member of. Instead of `None` , you may specify `'**'` .

`connect('SYS1')` returns 1 index level with user IDs. `connect(None,'IBMUSER')` or `connect(userid='IBMUSER')` returns 1 index level with group names.

You can find all entries in `.users` that have a group connection to SYSPROG as follows:

```
r.users.loc[r.users.USBD_NAME.isin(r.connect('SYSPROG').index)]
```

or

```
r.users.query("_NAME in @r.connect('SYSPROG').index")
```

These forms use the index structure of `.connect` , rather than the data, giving better speed. The 2nd example references the index field `_NAME` rather than the data column `USBD_NAME` .

dataset (*profile = None* , *option = None*) → ProfileFrame
> data frame with 1 record from `.datasets` when a profile is found, fitting the parameters exactly, or an empty frame.
>
> Example
>
> ```
> r.dataset('SYS1.*.**')
> ```
>
> To show all dataset profiles starting with SYS1 use:
>
> ```
> r.datasets.find('SYS1.**')
> ```
>
> To show the dataset profile covering SYS1.PARMLIB use:
>
> ```
> r.datasets.match('SYS1.PARMLIB')
> ```
>
> To find the access control list (acl) of profiles, use the `.acl()` method on any of these selections, e.g.:
>
> ```
> r.dataset('SYS1.*.**').acl()
> ```

datasetPermit (*profile = None* , *id = None* , *access = None* , *option = None*) → ProfileFrame
> data frame with records from `.datasetAccess` , fitting the parameters exactly, or an empty frame

Example

```
r.datasetPermit('SYS1.*.**', None, 'UPDATE')
```

This shows all IDs with update access on the `SYS1.*.**` profile (if this exists). To show entries from all dataset profiles starting with SYS1 use:

```
r.datasetAccess.find('SYS1.**', '**', 'UPDATE')
```

or

```
r.datasets.find('SYS1.**').acl(access='UPDATE')
```

datasetConditionalPermit (*profile = None* , *id = None* , *access = None* , *option = None*) → ProfileFrame

data frame with records from `.datasetConditionalAccess` , fitting the parameters exactly, or an empty frame.

Example

```
r.datasetConditionalPermit('SYS1.*.**', None, 'UPDATE')
```

To show entries from all conditional permits for `ID(*)` use:

```
r.datasetConditionalAccess.find('**', '*', '**')
```

general (*resclass = None* , *profile = None* , *option = None*) → ProfileFrame

data frame with profile(s) from `.generals` fitting the parameters exactly, or an empty frame.

Example

```
r.general('FACILITY', 'BPX.**')
```

If one of the parameters is written as `None` or `'**'` , or the second parameter is omitted, all profiles matching the specified parameter are shown:

```
r.general('UNIXPRIV')
```

To show the general resource profile controlling dynamic superuser, use:

```
r.general('FACILITY').match('BPX.SUPERUSER')
```

To show more general resource profiles relevant to z/OS UNIX use:

```
r.generals.find('FACILITY', 'BPX.**')
```

generalPermit (*resclass = None* , *profile = None* , *id = None* , *access = None* , *option = None*) → ProfileFrame

data frame with records from `.generalAccess` , fitting the parameters exactly, or an empty frame.

Example

```
r.generalPermit('UNIXPRIV', None, None, 'UPDATE')
```

This shows all IDs with update access on the any UNIXPRIV profile (if this exists). To show entries from all TCICSTRN profiles starting with CICSP use:

```
r.generalAccess.find('TCICSTRN', 'CICSP*')
```

generalConditionalPermit (*resclass = None* , *profile = None* , *id = None* , *access = None* , *option = None*) → ProfileFrame

data frame with records from `.generalConditionalAccess` fitting the parameters exactly, or an empty frame.

Example

```
r.generalConditionalPermit('FACILITY')
```

To show entries from all conditional permits for `ID(*)` use one of the following:

```
r.generalConditionalPermit('**', '**', '*', '**')
```

```
r.generalConditionalPermit(None, None, '*', None)
```

```
r.generalConditionalAccess.find(None, None, '*', None)
```

```
r.generalConditionalAccess.find(None, None, re.compile('\*'), None)
```

Selection Methods to use on DataFrames

The data tables described in Record types and properties present many entries. To find entries you can use standard pandas methods, or one of the methods specific for the RACF tables, relying on the index structure of these DataFrames. The result of these selections is another DataFrame.

In the examples below, `r` is a RACF object created from Parsing .

Selection Methods

The data tables are designed with an index to allow fast access and merging, but also provide easy to use selection capabilities. Depending on the entity type or RACF attribute stored in the table, one or more fields are assigned as index fields. These same index fields, in the same order, can be used with the following selection methods.

.find(*mask* , *mask* , ...)

The `.find` method emulates RACF generic filtering as implemented by the `SEARCH FILTER( )` TSO command. The generic characters `*` and `%` in these masks apply to the values in the RACF fields, ignoring the meaning of those characters in the profile field.

However, `'*'` looks for a single `*` in the field, such as with `ID(*)` . A value of `'**'` or `None` in a parameter means that any value is acceptable.

For backward compatibility, you can use `.gfilter( )` instead of `.find( )` to search for masks.

To select all data set profiles that start with SYS, you write

```
>>> r.datasets.find('SYS*.**')
SYSCSF.*.**
SYSHSM.**
SYS1.BRODCAST
SYS1.DFQP*
SYS1.DFQ*
SYS1.HRF*
SYS1.V%%%%%.**
SYS1.**.PAGE
```

```
SYS1.**
SYS2.RACFDS
SYS2.RACFDS.BACKUP
SYS2.**
```

For general resource profiles, you specify the class name and the profile key, as literals or patterns, for example

```
# all FACILITY profiles starting with BPX
r.generals.find('FACILITY', 'BPX.**')

# all general resource profiles starting with BPX
r.generals.find('**', 'BPX.**')

# all UNIXPRIV profiles
r.generals.find('UNIXPRIV')
```

For PERMITs, the ID and ACCESS values are available for selection too:

```
# dataset profiles where IBMUSER is permitted
r.datasetAccess.find('**', 'IBMUSER')

# IDs with UPDATE PERMIT on a SYS1 dataset profile
r.datasetAccess.find('SYS1.**', None, 'UPDATE')

# dataset where ID(*) has conditional access
r.datasetConditionalAccess.find(None, '*')

# UPDATE on a UNIXPRIV profile
r.generalAccesss.find('UNIXPRIV', '**', '**', 'UPDATE')
```

Selections can also contain a list of (literal) values:

```
# READ or UPDATE on a UNIXPRIV profile
r.generalAccesss.find('UNIXPRIV', '**', '**', ['READ','UPDATE'])
```

For group and user profiles, only one parameter is needed. Two parameters can be given for connect information:

```
r.groups.find('CSF*')

r.users.find('IBM*')

# users connected to SYS1, SYS2, etc.
r.connectData.find('SYS%')
```

```
# groups connected to PROD user IDs
r.connectData.find('**', '%%%%PROD')
```

Note: to check the index names defined in a DataFrame, use `.index.names`

```
>>> r.STDATA.index.names
FrozenList(['_CLASS_NAME', '_NAME'])
```

If there is no matching value, an empty DataFrame will be produced.

.find(re.compile(*regex*), ...)

The `.find` method supports regex patterns by accepting a pattern object. Remember that `*` and `.` in these patterns have a special significance, so prefix them with `\` if you want to search for `*` and `.` in the RACF fields.

```
import re

# SYS1 and SYS2 profiles
r.datasets.find(re.compile('SYS[12]\..*'))
```

or

```
from re import compile as R_

# dataset where ID(*) has conditional access
r.datasetConditionalAccess.find(None, R_('\*'))
```

The `.rfilter` method is provided for backward compatibility, it interprets the index patterns as regex strings. Internally, it also uses re.match().

```
# SYS1 and SYS2 profiles
r.datasets.rfilter('SYS[12]\..*')

# dataset where ID(*) has conditional access
r.datasetConditionalAccess.rfilter(None, '\*')

# user IDs with ADM anywhere
r.users.rfilter('.*ADM')

# groups ending in USER
r.groups.rfilter('\S+USER$')
```

.find(COLUMN = value , ...)

`.find( )` can be used to select entries through the value of a data field. Specify the column name with or without the table prefix, use a single `=` sign, and specify the selection value in quotes, unless you need to search for an integer or float value:

```
# special users with revoked status
r.users.find(SPECIAL='YES').find(REVOKE='YES')
```

Tests can also be combined, in which case both criteria must match:

```
# permit ID(SYS1) ACCESS(ALTER)
r.datasetAccess.find(DSACC_AUTH_ID='SYS1', DSACC_ACCESS='ALTER')
```

Selection on index fields and test on data fields can be combined:

```
# SYS1 data sets with UACC(UPDATE)
r.datasets.find('SYS1.**', UACC='UPDATE')
```

A list of values can be specified as a list:

```
# ID(*) with excessive access
r.datasetAccess.find(AUTH_ID='*',ACCESS=['UPDATE','CONTROL','ALTER'])

# permits for users with OPERATIONS
r.datasetAccess.find(AUTH_ID=r.operations.index)
```

.find(alias = value , ...)

Depending on the input table, alias names may be defined for specific columns. For access list related tables, `permits` is available to select permit entries for the user ID(s) and any of their connect groups:

```
# dataset permissions of CICSPRxx users
r.datasetAccess.find(permits='CICSPR*')
```

This offers a more efficient selection than `.acl(explode=True).find(user='CICSPR*')` or can be used as a pre-selection.

.skip(*mask* , ... , *COLUMN = value* , ...)

`.skip( )` excludes entries from further processing. The same parameters are supported as with `.find( )` :

```
# special users with revoked status, except IBMUSER
r.users.find(SPECIAL='YES', REVOKE='YES').skip('IBMUSER')

# profiles that do not have UACC=NONE, except the user catalogs
r.datasets.skip(UACC='NONE').skip('UCAT.**')
```

.match(*name*)

`match( )` finds the best fitting profile for a name, or a list of names:

```
# profile covering SYS1.PARMLIB
r.datasets.match('SYS1.PARMLIB')

# profile covering SYS1.PARMLIB, list access list
r.datasets.match('SYS1.PARMLIB').acl()

# profile covering BPX.SUPERUSER and IRR.PWRESET
r.generals.find('FACILITY').match(['BPX.SUPERUSER','IRR.PWRESET'])

Specify optional keyword ``show_resource=True`` to add a column
``RESOURCE`` to the ProfileFrame, containing the matched data set or
resource name::

# profile covering APF libraries
r.datasets.match(['SYS1.LINKLIB', 'SYS1.SVCLIB','USER.APFLOAD'],
show_resource=True)
```

Selection method syntax

ProfileFrame. find (** selection* , *** kwds*)

Search profiles using GENERIC pattern on the index fields.

selection can be one or more values, corresponding to index levels of the df. in addition(!), specify field names via an alias keyword or column name:

```
r.datasets.find("SYS1.**",UACC="ALTER")
```

specify regex using `re.compile` :

```
r.datasets.find(re.compile(r'SYS[12]\..*') )
```

ProfileFrame. skip (*selection* , ** *kwds*)

Exclude profiles using GENERIC pattern on the index fields.

selection can be one or more values, corresponding to index levels of the df alternatively, specify field names via an alias keyword or column name:

```
r.datasets.skip(DSBD_UACC="NONE")
```

FrameFilter. match (*selection* , *show_resource = False*)

dataset or general resource related records that match a given dataset name or resource.

Parameters :

· ***selection** – for dataset Frames: a dataset name. for general Frames: a resource name, or a class and a resource name. Each of these can be a str, or a list of str.

· **show_resource** (*bool*) – True: add a column with the resource name in the output Frame

Returns :

ProfileFrame with 0 or 1 entries for one resource, several if a list of resources is given

Example:

```
r.datasets.match('SYS1.PROCLIB')

r.datasets.match(['SYS1.PARMLIB','SYS1.PROCLIB'],
show_resource=True)

r.generals.match('FACILITY', 'BPX.SUPERUSER')

r.generals.find('FACILITY',match='BPX.SUPERUSER')
```

If you have a list of resource names, you can feed this into `match()` to obtain a ProfileFrame with a matching profile for each name. Next you concatenate these into one ProfileFrame and remove any duplicate profiles:

```python
resourceList = ['SYS1.PARMLIB','SYS1.PROCLIB']

profileList = r.datasets.match(resourceList)
```

or:

```python
profileList = pd.concat(
   [r.datasets.match(rname) for rname in resourceList]
                        ).drop_duplicates()
```

or:

```python
rlist = pd.DataFrame(resourceList, columns=['dsn'])

profileList = pd.concat(
        list(rlist.dsn.apply(r.datasets.match))
                        ).drop_duplicates()
```

and apply any of the methods on this profileList, such as:

```python
profileList.acl(resolve=True, allows='UPDATE')
```

Note: the resource name is not included in the output of acl(), so you should specify similar resources in the selection.

ProfileFrame. stripPrefix (*deep = False* , *prefix = None* , *setprefix = None*)
    remove table prefix from column names, for shorter expressions

    Parameters :

    · **deep** (*bool*) – shallow only changes column names in the returned value, deep=True changes the ProfileFrame.

    · **prefix** (*str*) – specified the prefix to remove if df._fieldPrefix is unavailable.

    · **setprefix** (*str*) – restores _fieldPrefix in the ProfileFrame if it was removed by .merge.

    Save typing with the query() function:

```python
r.datasets.stripPrefix().query("UACC==['CONTROL','ALTER']")
```

Deprecated method syntax

ProfileFrame. gfilter (*selection* , **kwds*)

    Search profiles using GENERIC pattern on the index fields.

    selection can be one or more values, corresponding to index levels of the df

    use `find()` for more options

ProfileFrame. rfilter (*selection* , **kwds*)

    Search profiles using refex on the index fields.

    selection can be one or more values, corresponding to index levels of the df

    use `find(re.compile('pattern'))` for more options

Pandas Methods

Data tables can also be processed with standard methods documented for pandas .

.loc[*value* , *value* , ...]

The exact value is looked up in the index field(s). This method is very fast, but an ugly `KeyError` is issued when there is no exact match. `.find` also uses the index fields, but suppresses the `KeyError` .

If there is one match, the result is given in a Series. To ensure the result is passed back as a DataFrame, you can double up the square brackets.

```
>>> r.users.loc['IBMUSER']
... Series object

>>> r.users.loc[['IBMUSER']]
... DataFrame
```

If the data table has more than one index field, and only one value is given in `.loc[ ]` , a DataFrame is produced with all entries for the value given.

```
>>> r.STDATA.loc['STARTED']
... DataFrame

>>> r.STDATA.loc['STARTED','ASCH.*']
```

```
GRST_RECORD_TYPE        0540
GRST_NAME               ASCH.*
GRST_CLASS_NAME      STARTED
GRST_USER_ID          START2
GRST_GROUP_ID
GRST_TRUSTED               NO
GRST_PRIVILEGED           NO
GRST_TRACE                NO
Name: (STARTED, ASCH.*), dtype: object

>>> r.STDATA.loc[[('STARTED','ASCH.*')]]
... DataFrame
```

By design, you specify index values as literals from the first level up, as in the previous examples. However, if you have to search the table for a value on, say, the third level and show any values found on the first two levels, you cannot just type `None` in those levels. Instead, you can use a "select anything" generator, enclose all selections in parentheses, and ensure that this tuple only acts on `axis=0` by adding a comma at the end. This is how you would find all permits to ID(*) in general resource profiles:

```
r.generalAccess.loc[(slice(None),slice(None),'*'),]
```

This is exactly what `.find('**','**','*')` would do, but more like a RACF person thinks.

Note:

> · .loc uses square brackets to specify the index value(s).

> · if a table has more than one index field, you may specify one or several, as long as they are in the right order.

> · if a table has more than one index field and you use the double brackets method, specify the index values as a tuple.

.loc[*bit array*]

The bit array variant of `.loc[ ]` can be used to search any of the fields in the table. The field names must be qualified with the table name, like so:

```
# IBM anywhere in the programmer name field
r.users.loc[ r.users.USBD_PROGRAMMER.str.contains('IBM') ]

# trusted and privileged started tasks
r.STDATA.loc[ (r.STDATA.GRST_TRUSTED=='YES')
            | (r.STDATA.GRST_PRIVILEGED=='YES') ]
```

```
# permits given to user IDs
r.datasetAccess.loc[
r.datasetAccess.DSACC_AUTH_ID.isin(r.users.index) ]

# orphan permits
r.datasetAccess.loc[
    ~ ( r.datasetAccess.DSACC_AUTH_ID.isin(r.users.index)
      | r.datasetAccess.DSACC_AUTH_ID.isin(r.groups.index)
      | (r.datasetAccess.DSACC_AUTH_ID=='*')
      )
]

# another way to write this, bypassing the issue with priority of ==
r.datasetAccess.loc[
    ~ ( r.datasetAccess.DSACC_AUTH_ID.isin(r.users.index)
      | r.datasetAccess.DSACC_AUTH_ID.isin(r.groups.index)
      | r.datasetAccess.DSACC_AUTH_ID.eq('*')
      )
]
```

The evaluations within the loc[ ] indexer are executed on all rows of the DataFrame, so for very large DataFrames, the number of comparisons may be … large. In such cases, the number of evaluations may be reduced by creating ever-smaller, temporary tables, like so:

```
orphans =
r.datasetAccess.loc[~r.datasetAccess.DSACC_AUTH_ID.isin(r.groups.index)]
orphans = orphans.loc[~orphans.DSACC_AUTH_ID.isin(r.users.index)]
orphans = orphans.loc[orphans.DSACC_AUTH_ID.ne('*')]
```

Creating the temporary DataFrame does not really copy the data, but only pointers to the data, so the benefits may outweigh the cost of the assignment.

Note:

- .loc uses square brackets to specify the selection.

- yes, you have to enter the full names of the data table inside the brackets.

- use `r.users.columns` to find the name of the columns in a table `r.users` .

- .loc[ ] with one array is somewhat intuitive, with two or more arrays you should use more parentheses rather than less, for example, around each comparison (==), and around the groups combined with the logical operators `&` , `|` and `~` . This is because these logical operators on vector data (arrays) have a higher priority than the comparison (==, !=, >, <) operators.

.query(*query string*)

The `.query` method makes it easier to search for records with values in specific fields, but documentation about the detailed syntax is hard to find. Here are some examples and some more . Also, you must write your query with two levels of quotes, one to enclose the query and another to specify literal strings. At least you do not have to refer to the table name in the query.

Like most methods, the result of one `.query()` can be passed (or chained) into another. The `\` serves as a continuation mark, like `,` in JCL and Rexx.

```
# privileged users
r.users.query("USBD_SPECIAL=='YES' or USBD_OPER=='YES'" +
               " or USBD_AUDITOR=='YES' or USBD_ROAUDIT=='YES'")\
       .query("USBD_REVOKE=='YES'")

# datasets with UACC>READ
r.datasets.query("DSBD_UACC==['UPDATE','CONTROL','ALTER']")
```

You can also correlate fields in one table with entries in another table.

```
# system special user forgot to remove themselves from OWNER( )
r.datasets.query("DSBD_OWNER_ID in @r.specials.index")
```

You can find all entries in .users that have a group connection to SYSPROG as follows. This references the user ID in index field `r.users._NAME` with the IDs connected to SYSPROG via the index:

```
r.users.query("_NAME in @r.connect('SYSPROG').index")
```

Query gives us access to the index field in the table, so we don't have to remember it's called _NAME:

```
r.users.query("index in @r.connect('SYSPROG').index")
```

You can also chain operators, for example to select the class of profiles first, considering that index based .loc[] is very fast and chaining it before query() drastically reduces the number entries query() has to test.

```
# conditional permission for operator commands from (SDSF etc)
console
```

```
r.generalConditionalAccess.loc['OPERCMDS']\
                          .query("GRCACC_CATYPE=='CONSOLE'")
```

With the pyracf `find()` method, this would be written as:

```
r.generalConditionalAccess.find('OPERCMDS',CATYPE='CONSOLE')
```

or as:

```
r.generalConditionalAccess.find('OPERCMDS').find(CATYPE='CONSOLE')
```

Output specification

The data tables described in Record types and properties contain many entries, and each entry will have many columns. To find entries you can use standard pandas methods, or one of the methods specific for the RACF tables, described in Selection Methods . This chapter shows how to customize output of the entries.

A DataFrame is a pandas object that is similar to a list of dictionaries (dicts). In fact, that is exactly how pyRACF feeds the RACF profile data into pandas. A row in the DataFrame is a list of (named) fields, a column in the DataFrame has a name, and many values.

In a traditional DataFrame the rows are numbered, but pyRACF assigns the value of one through four columns as *index* values. In the resulting `ProfileFrames` , the rows are identified by the *profile key* for groups, users and data sets, *class* and *profile* for general resource profiles, *group* and *user ID* for connectData. For access list entries, *access list ID* and *access level* are also used as *index* values.

These index values are printed in **bold** in front of the *data values* of the row. The will (all) be printed, even when you select only a few columns of the ProfileFrame.

Selecting columns for output

A column in a DataFrame is similar to an entry in a dictionary (dict), in the sense that the column has a name and how you access the column. If you wanted to print the 'brand' entry of your car dict, you would write car['brand']. This is how you would print the name of all user IDs:

```
>>> r.users['USBD_PROGRAMMER']

_NAME
irrcerta          CERTAUTH Anchor
irrmulti          Criteria Anchor
irrsitec              SITE Anchor
ADCDA                      ADCDA
ADCDB                      ADCDB
```

However, if you wanted to print 2 or more columns, pandas expects a list of field names, instead of one name:

```
>>>
r.users[['USBD_PROGRAMMER','USBD_LASTJOB_DATE','USBD_LASTJOB_TIME']]
```

```
              USBD_PROGRAMMER      USBD_LASTJOB_DATE    USBD_LASTJOB_TIME
_NAME
irrcerta      CERTAUTH Anchor
irrmulti      Criteria Anchor
irrsitec      SITE Anchor
ADCDA         ADCDA                2010-05-03           16:25:32
ADCDB         ADCDB                2009-12-14           16:53:49
```

You will notice a more austere layout with 1 column than with several columns. In fact, with 1 column output, the result is a `Series` , whereas with multiple columns, the result is a `DataFrame` .

How do we know these column names?

- use the columns attribute of the DataFrame `r.users.columns` , or

- look at the IBM documentation

pyRACF supports all the field names for all the record types documented in this list, with the same documented column names. For consistency and support this is great, but you may get fed-up with typing the identical field prefix for all the columns. Don't worry, there is a fix: the `stripPrefix` method is available on `ProfileFrames` , it removes the prefix in its return value:

```
>>> r.users.stripPrefix()
[['PROGRAMMER','LASTJOB_DATE','LASTJOB_TIME']]

              PROGRAMMER           LASTJOB_DATE     LASTJOB_TIME
_NAME
irrcerta      CERTAUTH Anchor
irrmulti      Criteria Anchor
irrsitec      SITE Anchor
ADCDA         ADCDA                2010-05-03       16:25:32
ADCDB         ADCDB                2009-12-14       16:53:49
```

What about the field name list, do you worry about counting quotes and commas? Well, it is a list, and you can easily create a list from a string using the split() method, and no need for double brackets either:

```
>>> r.users.stripPrefix()[ 'PROGRAMMER LASTJOB_DATE
LASTJOB_TIME'.split() ]
```

Just remember, split() is a method that works on the str value. And [ ] indexing works on a value, or on the result of a method, so don't add a `.` inbetween.

Look at the default layout for dataset profiles:

```
>>> r.datasets

    DSBD_RECORD_TYPE    DSBD_NAME    DSBD_VOL    DSBD_GENERIC
DSBD_CREATE_DATE    DSBD_OWNER_ID    DSBD_LASTREF_DATE
_NAME
ACEUSER.**      0400    ACEUSER.**                YES    2021-01-08
SYS1    2021-01-08  2021-01-08  00000   00000   ...
ACEV2.**        0400    ACEV2.**                  YES    2021-01-18
SYS1    2021-01-18  2021-01-18  00000   00000   ...
ADBC10.**       0400    ADBC10.**                 YES    2022-01-04
SYS1    2022-01-04  2022-01-04  00000   00000   ...
ADCD.S0W1.**    0400    ADCD.S0W1.**              YES    2014-02-18
SYS1    2014-02-18  2014-02-18  00000   00000   ...
ADCD.**         0400    ADCD.**                   YES    2012-01-10
NOTTHERE    2012-01-10  2012-01-10  00000   00000   ...
```

You cannot even see the UACC field, so what about:

```
>>> r.datasets.stripPrefix()['UACC IDSTAR_ACCESS ALL_USER_ACCESS
OWNER_ID'.split()]

             UACC      IDSTAR_ACCESS    ALL_USER_ACCESS      OWNER_ID
_NAME
ACEUSER.**      NONE                       NONE            SYS1
ACEV2.**        NONE   UPDATE              UPDATE          SYS1
ADBC10.**       NONE                       NONE            SYS1
ADCD.S0W1.**    NONE                       NONE            SYS1
ADCD.**         READ                       READ            NOTTHERE
```

You can combine these output specifications with selection methods:

```
r.datasets.find(ALL_USER_ACCESS='READ UPDATE CONTROL ALTER'.split()) \
.stripPrefix()['UACC IDSTAR_ACCESS ALL_USER_ACCESS OWNER_ID'.split()]
```

or

```
r.datasets.skip(ALL_USER_ACCESS='NONE') \
.stripPrefix()['UACC IDSTAR_ACCESS ALL_USER_ACCESS OWNER_ID'.split()]
```

ProfileFrame. stripPrefix (*deep = False* , *prefix = None* , *setprefix = None*)
     remove table prefix from column names, for shorter expressions

     Parameters :

       · **deep** (*bool*) – shallow only changes column names in the
         returned value, deep=True changes the ProfileFrame.

- **prefix** (*str*) – specified the prefix to remove if df._fieldPrefix is unavailable.

- **setprefix** (*str*) – restores _fieldPrefix in the ProfileFrame if it was removed by .merge.

Save typing with the query() function:

```
r.datasets.stripPrefix().query("UACC==['CONTROL','ALTER']")
```

Data presentation methods

.acl( )

The `.acl` method can be used on DataFrames with dataset and general resource profile, and on the corresponding access frames, to present various views of the access controls defined in these profiles.

When `.acl` is used on `.datasets` or `.generals` , normal and conditional access information is combined in the output. When `.acl` is used on one of the access frame, `.acl` shows just this data.

`.acl` returns a DataFrame without the prefixes of the originating frames.

```
>>> r.datasets.find('SYS1.**').acl()
          NAME   VOL USER_ID AUTH_ID ACCESS
---------------------------------------------
        SYS1.**       -group-    SYS1  ALTER
        SYS1.**         SPROG   SPROG  ALTER
        SYS1.**         TCPIP   TCPIP   READ
   SYS1.**.PAGE       -group-    SYS1  ALTER
  SYS1.BRODCAST             *       *   READ
```

The default layout shows *permits* much like the output of LISTDSD, except a column `USER_ID` is added. This contains the word `-group-` if the `AUTH_ID` was found in `r.groups` .

```
# user IDs with access on SYS1.PARMLIB (if this profile exists)
r.dataset('SYS1.PARMLIB').acl(resolve=True)

# permits with UPDATE on any SYS1 dataset profile
r.datasets.find('SYS1.**').acl(access='UPDATE')

# permits with UPDATE, CONTROL or ALTER on any SYS1 dataset profile
```

```
r.datasets.find('SYS1.**').acl(allows='UPDATE')

# users that can make changes to SYS1 datasets
r.datasets.find('SYS1.**').acl(allows='UPDATE',resolve=True)
```

To filter the output of `.acl()` you can chain `.query()` or `find()` , referencing the column names like so:

```
# access scope of IBMUSER in SYS1 data sets
r.datasets.find('SYS1.**')\
        .acl(resolve=True)\
        .query("USER_ID=='IBMUSER'")


# access scope of IBMUSER in SYS1 data sets
r.datasets.find('SYS1.**')\
        .acl(resolve=True)\
        .find(user='IBMUSER')
```

.acl( ) syntax

ProfileFrame. acl (*permits = True* , *explode = False* , *resolve = False* , *admin = False* , *access = None* , *allows = None* , *sort = 'profile'*)

> transform the {dataset,general}[Conditional]Access ProfileFrame into an access control list Frame

> Parameters :

> · **permits** (*bool*) – True: show normal ACL (with the groups identified as `-group-` in the USER_ID column).

> · **explode** (*bool*) – True: replace each groups with the users connected to the group (in the USER_ID column). A user ID may occur several times in USER_ID with various ACCESS levels.

> · **resolve** (*bool*) – True: show user specific permit, or the highest group permit for each user.

> · **admin** (*bool*) – True: add the users that have ability to change the profile or the groups on the ACL (in the ADMIN_ID column), VIA identifies the group name, AUTHORITY the RACF privilege involved.

- **access** (*str*) – show entries that are equal to the access
  level specified, e.g., access='CONTROL'.

- **allows** (*str*) – show entries that are higher or equal to the
  access level specified, e.g., allows='UPDATE'.

- **sort** (*str*) – sort the resulting output by column: user,
  access, id, admin, profile.

AclFrame. find (** selection* , *** kwds* **)**

  Search acl entries using GENERIC pattern on the data fields.

  selection can be one or more values, corresponding to data columns of the df. alternatively
  specify the field names via an alias keyword (user, auth, id or access) or column name in
  upper case:

```
r.datasets.acl().find(user="IBM*")
```

  specify regex using `re.compile` :

```
r.datasets.acl().find( user=re.compile('(IBMUSER|SYS1)') )
```

AclFrame. skip (** selection* , *** kwds* **)**

  Exclude acl entries using GENERIC pattern on the data fields.

  selection can be one or more values, corresponding to data columns of the df. alternatively
  specify the field names via an alias keyword (user, auth, id or access) or column name in
  upper case:

```
r.datasets.acl().skip(USER_ID="IBMUSER", ACCESS='ALTER')
```

Printable and python object attributes

Group Structure Properties

PyRACF presents two views of the RACF group tree: the link between groups through superior groups and subgroups, and the link through OWNER fields. These properties can be printed in two formats, the underlying structure can also be accessed through a dictionary, with superior levels as dict keys and the lower level groups as lists.

```
>>> r.ownertree
{'ADCDMST': ['BLZCFG', 'BLZWRK', 'XACFG', 'XAGUESTG', 'XASRVG'],
 'IBMUSER': ['ADCD',
  'BLZGRP',
  'CEAGP',
  'CFZADMGP',
  'CFZSRVGP',
  'CFZUSRGP',
  'CIMGP',
  'IMWEB',
  'IYU',
  'IYU0',
  'IYU0RPAN',
  'IYU0RPAW',
  'IYU000',
  'IZUADMIN',
  'IZUNUSER',
  'IZUSECAD',
  'IZUUNGRP',
  'IZUUSER',
  'STCGROUP',
  'SYSCTLG',
  {'SYS1': ['DB2',
    {'WEBGRP': ['EMPLOYEE', 'EXTERNAL']},
    'ZOSCGRP',
    'ZOSUGRP']},
  'TEST',
  'TTY',
  'USERCAT',
  'ZWEADMIN',
  'ZWE100']}
```

.grouptree

The grouptree property starts with SYS1 and shows how subgroups descend from SYS1.

```
>>> print(r.grouptree)
SYS1
├ ADCD
├ BLZCFG
├ BLZGRP
├ BLZWRK
├ CEAGP
├ CFZADMGP
├ CFZSRVGP
├ CFZUSRGP
├ CIMGP
├ IMWEB
├ IYU
│ ├ IYU0
│ │ └ IYU000
│ ├ IYU0RPAN
│ └ IYU0RPAW
├ IZUADMIN
├ IZUNUSER
├ IZUSECAD
├ IZUUNGRP
├ IZUUSER
├ VSAMDSET
├ WEBGRP
│ ├ EMPLOYEE
│ └ EXTERNAL
├ ZWEADMIN
└ ZWE100
```

.ownertree

The ownertree property starts with IBMUSER and all other user IDs that are specified as OWNER of a group. It shows the groups that reference these owners, and groups that reference those groups through OWNER, etc.

This ownership structure is critical in understanding the scope of the group privileges: group special, group operations and group auditor.

Note, there may be several user IDs identified as starting point, these are referred to as "breaks in the ownership tree".

```
>>> print(r.ownertree)
ADCDMST
├ BLZCFG
├ BLZWRK
├ XACFG
├ XAGUESTG
└ XASRVG
IBMUSER
├ ADCD
├ BLZGRP
├ CEAGP
├ CFZADMGP
├ CFZSRVGP
├ CFZUSRGP
├ CIMGP
├ IMWEB
├ IYU
├ IYU0
├ IYU0RPAN
├ IYU0RPAW
├ IYU000
├ IZUADMIN
├ IZUNUSER
├ IZUSECAD
├ IZUUNGRP
├ IZUUSER
├ STCGROUP
├ SYSCTLG
├ SYS1
│  ├ DB2
│  ├ WEBGRP
│  │  ├ EMPLOYEE
│  │  └ EXTERNAL
│  ├ ZOSCGRP
│  └ ZOSUGRP
├ TEST
├ TTY
├ USERCAT
├ ZWEADMIN
└ ZWE100
```

.setformat(*format*)

The default format used for printing the group structure trees is similar to the Unix `tree` command, and uses unicode box drawing characters. If these characters prove difficult to process, an alternative format can be selected. Valid format names are simple and unix.

```
>>> r.grouptree.setformat('simple')
>>> print(r.grouptree)
SYS1
| ADCD
| BLZCFG
| BLZGRP
| BLZWRK
| CEAGP
| CFZADMGP
| CFZSRVGP
| CFZUSRGP
| CIMGP
| IMWEB
| IYU
| | IYU0
| | | IYU000
| | IYU0RPAN
| | IYU0RPAW
| IZUADMIN
| IZUNUSER
| IZUSECAD
| IZUUNGRP
| IZUUSER
| VSAMDSET
| WEBGRP
| | EMPLOYEE
| | EXTERNAL
| ZWEADMIN
| ZWE100
```

.format(*format*)

`.format()` returns the printable format of the group tree in a `str` , suitable for further processing. The default format is similar to the Unix `tree` command, and uses unicode box drawing characters. If these characters prove difficult to process, an alternative format can be selected. Valid format names are simple and unix.

```
>>> r.grouptree.format('simple')
SYS1
| ADCD
| BLZCFG
| BLZGRP
| BLZWRK
| CEAGP
| CFZADMGP
| CFZSRVGP
| CFZUSRGP
```

```
| CIMGP
| IMWEB
| IYU
| | IYU0
| | | IYU000
| | IYU0RPAN
| | IYU0RPAW
| IZUADMIN
| IZUNUSER
| IZUSECAD
| IZUUNGRP
| IZUUSER
| VSAMDSET
| WEBGRP
| | EMPLOYEE
| | EXTERNAL
| ZWEADMIN
| ZWE100
```

tree syntax

class pyracf.group_structure. GroupStructureTree (*df* , *linkup_field = 'GPBD_SUPGRP_ID'*)

    dict with group names starting from SYS1 (group tree) or from (multiple) user IDs (owner tree).

    Printing these objects, the tree will be formatted as Unix tree (default, or after .setformat('unix') or with mainframe characters (after .setformat('simple').

    format (*format = 'unix'*)

        return printable tree

        Parameters :

        format – control character set to use in the tree representation. 'unix' for smart looking box characters, 'simple' for vertical bar and -

        Returns :

        printable string

    setformat (*format = 'unix'*)

        set default format for next print

Status Properties

.status

The `.status` property returns a dict with the current state of the class object.

```
>>> r.status
{
    'status': current_state,
    'input-lines': lines_in_irrdbu00_unload,
    'lines-read': lines_read,
    'lines-parsed': lines_parsed,
    'lines-per-second': lines_per_second,
    'parse-time': total_parse_time
}
```

The status field can have the following values:

Status	Meaning
Initial Object	RACF class has been instantiated, input-lines has a value
Error	Something went wrong
Still parsing your unload	pyRACF is busy parsing your input, lines-parsed shows progress
Optimizing tables	Parsing is done, pyRACf is now creating indexes etc. for faster lookups
Ready	All done, you can start querying.

.parsed(*table name*)

The `.parsed` method returns the number of records retrieved from the RACF input source, for a given table name or prefix. See Record types and properties for valid prefix values.

```
>>> r.parsed('USBD')
100
```

This way you can test if data was collected that would be needed for a report. Alternatively, you can use the `.empty` property of the DataFrame.

```
>>> r.users.empty
False
```

Verify RACF profiles using Rules

Administrative changes to RACF profiles may leave some fields in an undesirable state, be it through mistake or malice. Though finding such undesirable values takes time, they should be corrected quickly to minimize impact but also to address the cause of the mishap. Fixing long after the fact makes it difficult to find the cause since human memory is typically inaccurate.

Using ProfileFrames and python code it is possible to write algorithms that spot specific errors and inconsistencies, but such code can be complex and lengthy. Verifying for a multitude of possible inconsistencies results in more code than an auditor (or even a security analyst) can inspect.

The approach in the `RuleVerifier` package is to split the high level policy specification from the code that interacts with ProfileFrames. The policy specification states the desirable values of profile fields, the underlying *verifier* checks the individual Frames with reusable code.

Running a verification

Several common requirements for RACF administration have been combined in a module `profile_field_rules.py` , that can be easily run as a *default* policy. This produces a DataFrame with orphan permits, notify and owner values, permits in profiles that should not have any permits issued, incorrect users and groups used in STARTED profiles, etc.

You first create a RuleVerifier instance from a RACF object `r` like so:

```
from pyracf.rule_verify import RuleVerifier
v = RuleVerifier(r)
```

This can be shortened by relying on the `rules` property in the RACF object:

```
v = r.rules
```

The RuleVerifier instance needs a policy to run against the contents of the RACF objects. You can use the `load` method to add rules, domains, or pre-built modules with those. In the absence of parameters, `profile_field_rules.py` is loaded:

```
v.load()
```

This returns a modified RuleVerifier instance, but does not change the instance itself. You must either assign the returned value to save the modified instance or, more likely, execute one of the methods available:

```
>>> v.load().syntax_check()
      field    value    comment
0     rules    OK       No problem found in rules

>>> v.load().verify()
      CLASS        PROFILE                         FIELD_NAME
EXPECT      ACTUAL
RULE                                                     ID

  0    dataset      DSN710.ARCHLOG2.A0000008     DSACC_AUTH_ID
ACLID       DSN1MSTR    orphan permits

  1    dataset      DSN710.ARCHLOG2.B0000008     DSACC_AUTH_ID
ACLID       DSN1MSTR    orphan permits

181    JESSPOOL     &JESNODE.LEN*.*.*.*.*        id
USERQUAL    LEN*        2nd qualifier in JESSPOOL should be a user ID

182    SURROGAT     BPX.SRV.**                   user
USERQUAL    **          surrogate profiles must refer to user ID or
RA...

355    STARTED      DCEKERN.**                   GRST_GROUP_ID
GROUP       DCEGRP      orphans in STARTED profiles

356    STARTED      DCEPWDD.**                   GRST_GROUP_ID
GROUP       DCEGRP      orphans in STARTED profiles

357    STARTED      DCESECD.**                   GRST_GROUP_ID
GROUP       DCEGRP      orphans in STARTED profiles
```

`syntax_check` merely looks at the policy and checks that no illegal components are used. `verify` checks a number of ProfileFrames and lists (possible) problems in a DataFrame. Columns in the Frame are:

CLASS

The class of the general resource profile, or *user* , *group* or *dataset* .

PROFILE

The key of the profile where the issue was spotted.

FIELD_NAME

The name documented in the IBM Documentation . This also identifies the prefix of the ProfileFrame, or the table name (DSACC, GRST, etc).

EXPECT

> The `domain` name for the field, or the literal values, as stated in the `rule` . ACLID is a domain that includes all user IDs, group names and * .

ACTUAL

> The actual value found in the field, this would be the value that conflicts with the `rule` .

RULE

> Descriptive name of the `rule` .

ID

> Optional numeric or character identifier of the `rule` .

The DataFrame can be further manipulated using `find` and `skip` methods using the (uppercase) column headers, or (lowercase) aliases of the columns. For example, select all issues with JESPOOL and SURROGAT profiles, and save these in a comma separated file:

```
v.load().verify().find(resclass=['JESSPOOL','SURROGAT']).to_csv('/
tmp/issues_for_sysprog.csv')
```

Rules example

Rules are processed as a python dictionary, using the dictionary keys as directive and parameter names, and the dictionary values as criteria and parameters. To improve readability, you can use yaml to write and store rules, but keep in mind that yaml aggressively interprets parameter values as bool, int or float values, unless you add quotes around the value. See The yaml document from hell .

If the `load` method finds a `dict` type parameter, it uses the dict as a rules or domains value. If it receives a `str` type, it converts this from yaml to dict before using it. We will use yaml to illustrate the structure of rules.

Suppose we want to test the permits on data set and general resource profiles, to verify that the IDs (still) exist. We would process the dataset access (DSACC) and general resource access (GRACC) tables, test the value of DSACC_AUTH_ID and GRACC_AUTH_ID to see if these are (valid) Access Control List IDs (ACLID). Instead of writing the whole field name, we leave off the prefix because the remainder is the same in those two tables. The following would accomplish the *orphan permit* test:

```
testPermits = '''
permits must refer to existing users or groups:
  - [DSACC,GRACC]
  - test:
      field: AUTH_ID
```

```
        fit: ACLID
  '''
  v.load(rules=testPermits).verify()
```

The first line in the *multi-line string* contains the rule description, written as the key of the dict entry (that's what the `:` at the end of the line is for).

The value of the dict entry is a list (array). The first element of the list is either a str with the table name (field prefix), or a list of str when multiple tables should be processed. The other elements of the list describe test criteria: tests and selections that limit where the test should be performed.

In this example, the test applies to the *prefix* _AUTH_ID field and checks that the value *fits* the ACLID `domain`. See Domains example below for other domains.

We can also apply two rules in one verify:

```
  testAccess = '''
  dataset permits must refer to existing users or groups:
    - DSACC
    - test:
      field: AUTH_ID
      fit: ACLID

  no update access to data sets through UACC:
    - DSBD
    - test:
        field: UACC
        value: [NONE,READ]
  '''
  v.load(rules=testAccess).verify()
```

Each rule starts with the (key) rule description, followed by the table name. The rules apply to different tables. The second rule verifies that the UACC of dataset profiles does not exceed READ.

So far, the test commands were not preceded by selections, so they apply to all entries in the specified table. We expand the first rule with a similar restriction to access for `ID(*)`. This is accomplished by adding a new entry to the end of the list, this time with a find and a test directive:

```
  testAccess = '''
  access to data sets through permits:
    - DSACC
    - test:
        field: AUTH_ID
        fit: ACLID
    - find:
```

```
      field: AUTH_ID
      value: '*'
   test:
      field: ACCESS
      value: [NONE,READ]

no update access to data sets through UACC:
  - DSBD
  - test:
      field: UACC
      value: [NONE,READ]
'''
v.load(rules=testAccess).verify()
```

The first rule now contains two test criteria. The first applies to all DSACC entries, the second only to entries where AUTH_ID contains an asterisk. For these `ID(*)` entries, the same test is applied as to the UACC value.

Lets add a test for the WARNING flag in the profile (Basic Data):

```
testAccess = '''
access to data sets through permits:
  - DSACC
  - test:
      field: AUTH_ID
      fit: ACLID
  - find:
      field: AUTH_ID
      value: '*'
   test:
      field: ACCESS
      value: [NONE,READ]
no update access to data sets through UACC:
  - DSBD
  - test:
      - field: UACC
        value: [NONE,READ]
      - field: WARNING
        value: 'NO'
        rule: Warning mode must be disabled
'''
v.load(rules=testAccess).verify()
```

The test directive now contains a list of field-value criteria, so two fields are checked for each entry. This also demonstrate that rule descriptions can be specified at the rule **or** at the test level.

Finally, an optional directive `id` can be specified at the same level as the test directive, or in the field criteria:

```
testAccess = '''
access to data sets through permits:
  - DSACC
  - id: 1.1
    test:
      field: AUTH_ID
      fit: ACLID
  - id: 1.2
    find:
      field: AUTH_ID
      value: '*'
    test:
      field: ACCESS
      value: [NONE,READ]
no update access to data sets through UACC:
  - DSBD
  - id: 1.3
    test:
      - field: UACC
        value: [NONE,READ]
      - field: WARNING
        value: 'NO'
        id: 1.4
        rule: Warning mode must be disabled
'''
v.load(rules=testAccess).verify()
```

And you can filter the verify results using the `find` method, like so:

```
v.load(rules=testAccess).verify().find(ID=1.4)
```

Rules syntax

Rules are a dictionary (dict), the description of the rule is the key of a dict entry. Normally, yaml ignores entries with a duplicate description, however, RulesVerifier issues a warning and creates a unique key.

The entry value is a list, the first element of the list identifies the table or tables this rule works on. The table name is identified by the *Prefix* value shown in Record types and properties . The table name can also be a dynamic table created with the `save` directive.

Subsequent list elements are test criteria.

Test criteria are a dict. The keys of the criteria dict are referred to as directives. An output directive is required, that is `save` or `test` , all others are optional.

class, -class

Applies only to tables starting with GR, select or exclude entries of the specified general resource class. Patterns are not supported. Provides fast selection.

profile, -profile

Select or exclude profiles (keys) that match the generic pattern given. Provides fast selection.

match, -match

Select or exclude profiles that provide the best match with the given data set name or general resource name. For example:

```
match: SYS1.PROCLIB
```

or a list:

```
match:
  - SYS1.PROCLIB
  - SYS1.USER.PROCLIB
```

or:

```
class: FACILITY
match: BPX.SUPERUSER
```

If the match value contains parentheses, the value extracted from the corresponding qualifier in the profile key will be stored in a new field named by the string within the parentheses, and can be used in the `find` , `skip` and `test` directives. For example:

```
- class: SURROGAT
  match: (id).SUBMIT
  test:
    field: id
    fit: USERQUAL
```

find, skip

Select or exclude profiles using field names. These directives accept one field criterium, or several in a list. If more than one criterium is given, the criteria must all match, in other words, they act as AND conditions. For example, this excludes all permits to SYS1 with ALTER access:

```
skip:
  - field: AUTH_ID
    value: SYS1
  - field: ACCESS
    value: ALTER
```

If an OR condition is needed, you can specify additional `find` and `skip` directives with arbitrary characters after the 4 fixed letters, for example:

```
find_s:
  field: SPECIAL
  value: 'YES'
find_o:
  field: OPER
  value: 'YES'
find_a:
  field: AUDITOR
  value: 'YES'
find_roa:
  field: ROAUDIT
  value: 'YES'
```

Parameters in the `find` and `skip` criteria:

field

Field name, with or without prefix. You can specify field names from the current table, joined table, or dynamic fields from the `match` directive.

value

The value the field should have, or a list of values. Be careful to add quotes around YES, NO, ON, OFF, FAIL, FALSE and TRUE. Patterns are not supported. If `fit` and `value` are both specified, the field value matches if it is either in the domain, or it matches the value.

fit

The name of a domain entry, the current field value must be a member of the domain for `find`, or not for `skip`.

eq

Field name, with or without prefix, to compare with the value of `field` , for example, to select groups where owner and superior group are equal.

ne

Field name, with or without prefix, to compare with the value of `field` , for example, to select DSACC records where the user ID on the ACL is not the same as the hight level qualifier.

join

Retrieve additional data fields from another table. The target table will be accessed through its index. The table name is a *Prefix* , or a dynamic table name defined with the `save` directive. If the `on` parameter is omitted, a match with the current index value will be found, for example to add segment data to a base definitions table:

```
specials should not have root:
  - USBD
  - id: 101
    join: USOMVS
    find:
      field: SPECIAL
      value: 'YES'
    test:
      field: UID
      value: '0000000000'
      action: 'FAIL'

specials should not also have group special:
  - USBD
  - id: 102
    join:
      table: USCON
      how: inner
    find:
      field: SPECIAL
      value: 'YES'
    test:
      field: GRP_SPECIAL
      value: 'NO'

specials must be connected to RACFADM:
  - USBD
  - id: 103
    join:
      table: USCON
      how: left
    find:
```

```
              - field: SPECIAL
                value: 'YES'
              - field: GRP_ID
                value: RACFADM
          test:
              - field: GRP_SPECIAL
                value: 'NO'
              - field: USCON_REVOKE
                value: 'NO'
```

Parameters of the `join` directive:

> Name of the target table

or a dict with keys:

table

> Name of the target table.

on

> Field name in the current table to use for lookup in the target table. When omitted, the
> index field of the current table is used.

how

> Join method, 'left', 'right', 'outer', 'inner', or 'cross'. See pandas documentation for the use of
> join methods.

`join` can also reference a saved table, for example, to match user IDs in *permits* with a dynamic
list of user IDs, see below .

save

Save the result of the current selection as a local (within this verify() run) table name, so a
subsequent rule can refer to the saved results by name. All results of the directives in the current
rule are saved, except the `test` directive and the matched (dynamic) field values. Saving rule
results may reduce processing time, especially when the results were derived from a `match`
operation. `save` is denied if the rule reads from multiple tables.

An example where the APF data set names were used to select profiles, and the 3rd rule uses
these profiles to verify corresponding records with access list info. Note how field names in the
3rd rule include the table prefix to prevent name clashes:

```
APF library updates must be controlled:
- DSBD
- match:
      - SYS1.LINKLIB
```

```
            - TEST.APFLOAD
            - TEST.USERAPF
      save: APF_profiles
      test:
        - field: UACC
          value:
            - NONE
            - READ
        - field: IDSTAR_ACCESS
          value:
            - NONE
            - READ
        - field: WARNING
          value: 'NO'

 APF library updates must be logged:
 - APF_profiles
 - test:
      - field: AUDIT_LEVEL
        value: [ALL,SUCCESS]
      - field: AUDIT_OKQUAL
        value: [READ,UPDATE]

 APF library update must be limited to sysprogs:
 - APF_profiles
 - join: DSACC
   find:
      - field: DSACC_ACCESS
        value: [UPDATE,CONTROL,ALTER]
   test:
      field: DSACC_AUTH_ID
      value:
        - SYS1
        - SYSPROG
```

The saved table can also be used in a `join` directive, for example, to select entries from a list
built in a previous rule:

```
 Users with special:
 - USBD
 - find:
      field: SPECIAL
      value: 'YES'
   -profile: IBMUSER
   save: Special_users

 Specials should have no DATASET permit ALTER:
 - DSACC
 - join:
      table: Special_users
```

```
   on: AUTH_ID
   how: inner
 find:
   field: ACCESS
   value: ALTER
 test:
   field: AUTH_ID
   value: ''
```

test

Perform test on field values in the selected profiles. The directive requires one field criterium specifying the expected value(s), or several criteria in a list. If more than one criterium is given, the criteria must all match, in other words, they act as AND conditions.

Parameters in the `test` directive:

field

> Field name, with or without prefix. You can specify field names from the current table, joined table, or dynamic fields from the `match` directive.

value

> The value the field should have, or a list of values. Be careful to add quotes around YES, NO, ON, OFF, FAIL, FALSE and TRUE. Patterns are not supported. If `fit` and `value` are both specified, the field value matches if it is either in the domain, or it matches the value.

fit

> The name of a domain entry, the current field value must be a member of the domain for `find` , or not for `skip` .

eq

> Field name, with or without prefix, to compare with the value of `field` , for example, to check if owner and superior group are equal.

ne

> Field name, with or without prefix, to compare with the value of `field` , for example, to check if the user ID on the ACL is not the same as the hight level qualifier.

action

> Reverse the result of the field test by specifying action: 'FAILURE', 'FAIL', 'F', or 'V'.

id

The rule can be identified with a str, int or float value. This can be specified as a directive, or as a parameter in the `test` directive.

rule

An overriding rule description can be specified as a directive, or as a parameter in the `test` directive.

Domains example

Domains are processed as a python dictionary, using the dictionary keys as the domain name and the dictionary values as members of the domain. The value must be a list-like object. To improve readability, you can use yaml to write and store domains.

If the `load` method finds a `dict` type parameter, it uses the dict as a rules or domains value. If it receives a `str` type, it converts this from yaml to dict before using it.

The default policy module `profile_field_rules.py` introduces some useful domains:

USER
> List of all RACF defined user IDs.

GROUP
> List of all RACF defined group names.

ID
> The `union` of USER and GROUP, giving IDs that could be used as, for example, OWNER of a profile.

SPECIALID
> Special values that can be used in (some) PERMITs and profile qualifiers: `*` , `&RACUID` , and `&RACGRP` .

ACLID
> The `union` of ID and SPECIALID, providing a domain to use for verifying PERMITs.

RACFVARS
> The RACFVARS profile keys, e.g., `&RACLNDE` .

USERQUAL

The `union` of USER and RACFVARS, used to check profile qualifiers that should contain a (configuarable) user ID.

CATEGORY, SECLEVEL, SECLABEL

List of categories, security levels and security labels defined in their relevant general resource profiles.

DELETE

An empty domain, to ascertain a field is empty.

These domain names may be used with the `fit` parameter in `find` , `skip` and `test` directives, like so:

```
testNotify = '''
NONOTIFY should be used on all profiles:
  - [DSBD,GRBD]
  - test:
      field: NOTIFY_ID
      fit: DELETE
'''
v.load(rules=testNotify).verify()
```

or:

```
testNotify = '''
NOTIFY only works for user IDs that have not been deleted:
  - [DSBD,GRBD]
  - test:
      field: NOTIFY_ID
      fit: USER
'''
v.load(rules=testNotify).verify()
```

In addition to the pre-defined domains, you can add your own using list-like objects as values:

```
v.add_domains('''
CICS_REGIONS:
    - CICPRODA
    - CICPRODB
    - CICSTEST
''')

v.add_domains({'PROD_GROUPS': ['PRODA','PRODB','PRODCICS'],
               'TEST_GROUPS': ['TEST1','TEST2']})

v.add_domains({'SYS1': r.connect('SYS1').index})
```

```
v.add_domains({'omvs_root_group': r.connect(user='OMVSKERN').index})
```

The third example shows how the `connect` attribute can be used to populate a domain with user IDs, the last example retrieves group names from a user ID.

Note: the parameter for `add_domains` is a dict and replaces identically named entries in the domain map with no warning.

You can use the `get_domains` method to extract one or all domain entries from the verify instance:

```
v.get_domains()  # get a dict of all domains
v.get_domains('SECLABEL')  # get 1 domain in a list
```

Verify access controls on APF libraries

If you have a list of critical data set names, for example, APF libraries, you can find the corresponding profiles as follows, and create a domain of these critical profiles:

```
apfLibraries = ['SYS1.LINKLIB', 'TEST.APFLOAD', 'TEST.USERAPF']
apfProfiles = r.datasets.match(apfLibraries)
v.add_domains({'APF profiles': apfProfiles.index})
```

You can do the same with the built-in yaml support:

```
v.add_domains('''
APF libraries:
    - SYS1.LINKLIB
    - TEST.APFLOAD
    - TEST.USERAPF
''')
v.add_domains({'APF profiles': r.datasets.match(v.get_domains('APF
libraries')).index})
```

Next, you can use this domain to select dataset profiles and access list entries that *fit* the profiles in this domain, and apply tests:

```
v.load(rules='''
APF library update must be controlled and logged:
  - [DSBD]
  - find:
      - field: NAME
```

```
        fit: APF profiles
    test:
      - field: UACC
        value:
          - NONE
          - READ
      - field: WARNING
        value: 'NO'
      - field: AUDIT_LEVEL
        value: [ALL,SUCCESS]
      - field: AUDIT_OKQUAL
        value: [READ,UPDATE]

 APF library update must be limited to sysprogs:
   - [DSACC]
   - find:
       - field: NAME
         fit: APF profiles
       - field: ACCESS
         value: [UPDATE,CONTROL,ALTER]
     test:
       field: AUTH_ID
       value:
         - SYS1
         - SYSPROG
   ''')

 v.verify()
```

Identify orphans in access control lists

The default policy module `profile_field_rules.py` contains a rule to find orphans in dataset and general resource profiles. This verification can also be called stand-alone and with a reduced output frame, with the following code:

```python
from pyracf.rule_verify import RuleVerifier

orphans = RuleVerifier(r)\
   .load(rules = {'orphan permits':
                  (['DSACC','DSCACC','GRACC','GRCACC'],
                   {'test': {'field':'AUTH_ID', 'fit':'ACLID'}}) } )\
   .verify()\
   .drop(['FIELD_NAME','EXPECT','RULE','ID'],axis=1)\
   .rename({'ACTUAL':'AUTH_ID'},axis=1)\
   .set_index('AUTH_ID')
```

This produces a frame `orphans` by orphan ID, with class and profile key, ready to generate `PERMIT DELETE` commands.

Methods and classes for RuleVerifier

class pyracf.rule_verify. RuleFrame (*data = None* , *index : Axes | None = None* , *columns : Axes | None = None* , *dtype : Dtype | None = None* , *copy : bool | None = None*)

    Bases: `DataFrame` , `FrameFilter`

    Output of a verify() action

    find (** selection* , *** kwds*)

        Search rule results using GENERIC pattern on the data fields. selection can be one or more values, corresponding to data columns of the df.

        alternatively specify the field names via an alias keyword (resclass, profile, field, actual, found, expect, fit, value or id):

```
r.rules.load().verify().find(field='OWN*')
```

        specify selection as regex using re.compile:

```
r.rules.load().verify().find( field=re.compile('(OWNER|DFLTGRP)' )
```

    skip (** selection* , *** kwds*)

        Exclude rule results using GENERIC pattern on the data fields. selection can be one or more values, corresponding to data columns of the df

        alternatively specify the field names via an alias keyword (resclass, profile, field, actual, found, expect, fit, value or id):

```
r.rules.load().verify().skip(actual='SYS1')
```

class pyracf.rule_verify. RuleVerifier (*RACFobject*)

    Bases: `object`

verify fields in profiles against expected values, issues are returned in a df.

rules can be passed as a dict of [tuples or lists], and a dict with domains, via parameter, or as function result from external module. created from RACF object with the .rules property.

    load (*rules = None* , *domains = None* , *module = None* , *reset = False* , *defaultmodule = '.profile_field_rules'*)

        load rules + domains from yaml str, structure or from packaged module

Parameters :

- **rules** (*dict* , *str*) – dict of tuples or lists with test specifications, or yaml str field that expands into a dict of lists

- **domains** (*dict* , *str*) – one or more domain in a dict(name=[entries]), or in a yaml string

- **module** (*str*) – name of module that contains functions rules() and domains()

- **defaultmodule** (*str*) – module name to be used if all parameters are omitted

- **reset** (*bool*) – clear rules, domains and module in RuleVerifier object, before loading new values

Returns :

the updated object

Return type :

RuleVerifier

Example:

```
r.rules.load(rules = {'test libraries':
   (['DSBD'],
    {'id': '101',
     'rule': 'Integrity of test libraries',
     'profile': 'TEST*.**',
     'test': [{'field':'UACC', 'value':['NONE','READ']},
             {'field':'WARNING', 'value':'NO'},
             {'field':'NOTIFY_ID', 'fit':'DELETE'}],
    }
   )
              }
        ).verify()
```

add_domains (*domains = None*)
    Add domains to the end of the domain list, from a dict or a yaml string value.

Parameters :

domains (*dict* , *str*) – one or more domains in a dict(name=[entries]), or in a yaml string

Returns :

The updated object

Return type :

RuleVerifier

Example:

```
v = r.rules.load()

v.add_domains({'PROD_GROUPS': ['PRODA','PRODB','PRODCICS'],
               'TEST_GROUPS': ['TEST1','TEST2']})

v.add_domains({'SYS1': r.connect('SYS1').index})
```

get_domains (*domains = None*)
Get domain definitions as a dict, or one entry as a list.

Parameters :

str (*domains*) – name of domain entry to return as list, or None to return all

Returns :

dict or list

Example:

```
v.get_domains() # all domains as a dict

v.get_domains('PROD_GROUPS') # one domain as a list
```

verify (*rules = None* , *domains = None* , *module = None* , *reset = False* , *id = True* , *verbose = False* , *syntax_check = None* , *optimize = 'rows cols'*) → RuleFrame
verify fields in profiles against the expected value, issues are returned in a df

Parameters :

- **id** (*bool*) – False: suppress ID column from the result
  frame. The values in this column are taken from the id
  property in rules

- **syntax_check** (*bool*) – deprecated

- **verbose** (*bool*) – True: print progress messages

- **optimize** (*str*) – cols to improve join speed, rows to
  use pre-selection

Returns :

Result object (RuleFrame)

Example:

```
r.rules.load().verify()
```

syntax_check (*confirm = True*) → RuleFrame

parse rules and domains, check for consistency and unknown directives, normalize
field names

specify confirm=False to suppress the message when all is OK

Parameters :

confirm (*bool*) – False if the success message should be
suppressed, so in automated testing the result frame
has .empty

Returns :

syntax messages (RuleFrame)

Example:

```
r.rules.load().syntax_check()

if r.rules.load().syntax_check(confirm=False).empty:
    print('No syntax errors in default policy')
```

pyracf

pyracf package

Submodules

pyracf.frame_filter module

class pyracf.frame_filter. FrameFilter

Bases: `object`

filter routines that select or exclude records from a the 3 DataFrames classes

match (** selection* , *show_resource = False*)

dataset or general resource related records that match a given dataset name or resource.

Parameters :

- ***selection** – for dataset Frames: a dataset name. for general Frames: a resource name, or a class and a resource name. Each of these can be a str, or a list of str.

- **show_resource** (*bool*) – True: add a column with the resource name in the output Frame

Returns :

ProfileFrame with 0 or 1 entries for one resource, several if a list of resources is given

Example:

```
r.datasets.match('SYS1.PROCLIB')
```

```
r.datasets.match(['SYS1.PARMLIB','SYS1.PROCLIB'],
show_resource=True)

r.generals.match('FACILITY', 'BPX.SUPERUSER')

r.generals.find('FACILITY',match='BPX.SUPERUSER')
```

If you have a list of resource names, you can feed this into `match()` to obtain a ProfileFrame with a matching profile for each name. Next you concatenate these into one ProfileFrame and remove any duplicate profiles:

```
resourceList = ['SYS1.PARMLIB','SYS1.PROCLIB']

profileList = r.datasets.match(resourceList)
```

or:

```
profileList = pd.concat(
   [r.datasets.match(rname) for rname in resourceList]
                         ).drop_duplicates()
```

or:

```
rlist = pd.DataFrame(resourceList, columns=['dsn'])

profileList = pd.concat(
        list(rlist.dsn.apply(r.datasets.match))
                         ).drop_duplicates()
```

and apply any of the methods on this profileList, such as:

```
profileList.acl(resolve=True, allows='UPDATE')
```

Note: the resource name is not included in the output of acl(), so you should specify similar resources in the selection.

pyracf.getOffsets module

pyracf.group_structure module

class pyracf.group_structure. GroupStructureTree (*df* , *linkup_field = 'GPBD_SUPGRP_ID'*)

    Bases: `dict`

dict with group names starting from SYS1 (group tree) or from (multiple) user IDs (owner tree).

Printing these objects, the tree will be formatted as Unix tree (default, or after .setformat('unix') or with mainframe characters (after .setformat('simple').

format (*format = 'unix'*)

    return printable tree

    Parameters :

    format – control character set to use in the tree representation. 'unix' for smart looking box characters, 'simple' for vertical bar and -

    Returns :

    printable string

setformat (*format = 'unix'*)

    set default format for next print

unix_format (*branch = None* , *prefix = ''*)

    print groups, prefixed with box characters to show depth

simple_format (*branch = None* , *depth = 0*)

    print groups, prefixed with vertical bars to show depth

property **tree**

    deprecated, the dict is now the default return value of the object

pyracf.profile_field_rules module

rules for the pyracf.verify() service. This module is imported from load() on a rules object. load() expects a dict for domains and for rules, or a yaml str representing these objects.

functions:

domains: returns the sets of values that can be expected in profile fields and qualifiers. rules: specifies where these domain values should be expected.

pyracf.profile_field_rules. domains (*self* , *pd*)

generate a dict (or yaml str) with named lists of values

each domain entry contains a list, array or Series that will be used in .loc[field.isin( )] to verify valid values in profile fields. keys of the dict are only referenced in the corresponding rules, feel free to change /extend. self and pd are passed down to access data frames from caller.

pyracf.profile_field_rules. rules (*self* , *format = 'yaml'*)

generate a dict of lists/tuples, each with (list of) table ids, and one or more conditions.

key of the dict names the rule described in the dict entry.

each condition allows class, -class, profile, -profile, find, skip, match and test.

class, -class, profile and -profile are (currently) generic patterns, and select or skip profile/ segment/entries.

find and skip can be used to limit the number of rows to process

match extracts fields from the profile key, the capture name should be used in subsequent fields rules. match changes . to . and * to *, so for regex patterns you should use S and +? instead.

test verifies that the value occurs in one of the domains, or a (list of) literal(s).

id and rule document the test at the test level or at the field level within a test.

pyracf.profile_filter_keywords module

class pyracf.profile_filter_keywords. ProfileFilterKeywords

Bases: `object`

generation routines for keywords on find/skip.

Parameters :

- **df** (*ProfileFrame*) – frame to filter

- **kwd** (*str*) – (alias) keyword found in filter command

- **sel** (*str* , *list*) – selection value found in filter command

Returns :

list of field name and field values to use in loc[ ]

add (*entry*)
    Add alias to map.

    Parameters :

    · kwd (*str*) – alias name to use in find/skip on
      ProfileFrames

    · entry (*str* , *callable*) – processor of alias

    Example

    import pyracf from pyracf.profile_filter_keywords import ProfileFilterKeywords
    ProfileFilterKeywords.add('name', 'COLUMN_NAME')

pyracf.profile_frame module

class pyracf.profile_frame. AclFrame (*data = None* , *index : Axes | None = None* , *columns : Axes |*
None = None , *dtype : Dtype | None = None* , *copy : bool | None = None*)
    Bases: `DataFrame` , `FrameFilter`

    output of the .acl() method

find (** selection* , *** kwds*)
    Search acl entries using GENERIC pattern on the data fields.

    selection can be one or more values, corresponding to data columns of the df.
    alternatively specify the field names via an alias keyword (user, auth, id or access) or
    column name in upper case:

    ```
    r.datasets.acl().find(user="IBM*")
    ```

    specify regex using `re.compile` :

    ```
    r.datasets.acl().find( user=re.compile('(IBMUSER|SYS1)') )
    ```

skip (** selection* , *** kwds*)
    Exclude acl entries using GENERIC pattern on the data fields.

selection can be one or more values, corresponding to data columns of the df. alternatively specify the field names via an alias keyword (user, auth, id or access) or column name in upper case:

```
r.datasets.acl().skip(USER_ID="IBMUSER", ACCESS='ALTER')
```

class pyracf.profile_frame. ProfileFrame (*data = None* , *index : Axes | None = None* , *columns : Axes | None = None* , *dtype : Dtype | None = None* , *copy : bool | None = None*)

Bases: `DataFrame` , `FrameFilter` , `XlsWriter`

pandas frames with RACF profiles, the main properties that the RACF object provides

read_pickle ( )

to_pickle (*path*)

ensure RACFobject is not saved in pickle

find (** selection* , *** kwds*)

Search profiles using GENERIC pattern on the index fields.

selection can be one or more values, corresponding to index levels of the df. in addition(!), specify field names via an alias keyword or column name:

```
r.datasets.find("SYS1.**",UACC="ALTER")
```

specify regex using `re.compile` :

```
r.datasets.find(re.compile(r'SYS[12]\..*') )
```

skip (** selection* , *** kwds*)

Exclude profiles using GENERIC pattern on the index fields.

selection can be one or more values, corresponding to index levels of the df alternatively, specify field names via an alias keyword or column name:

```
r.datasets.skip(DSBD_UACC="NONE")
```

gfilter (** selection* , *** kwds*)

Search profiles using GENERIC pattern on the index fields.

selection can be one or more values, corresponding to index levels of the df

use `find()` for more options

rfilter (** selection* , *** kwds*)

Search profiles using refex on the index fields.

selection can be one or more values, corresponding to index levels of the df

use `find(re.compile('pattern'))` for more options

stripPrefix (*deep = False* **,** *prefix = None* **,** *setprefix = None* **)**
remove table prefix from column names, for shorter expressions

Parameters :

- **deep** (*bool*) – shallow only changes column names in the returned value, deep=True changes the ProfileFrame.

- **prefix** (*str*) – specified the prefix to remove if df._fieldPrefix is unavailable.

- **setprefix** (*str*) – restores _fieldPrefix in the ProfileFrame if it was removed by .merge.

Save typing with the query() function:

```
r.datasets.stripPrefix().query("UACC==['CONTROL','ALTER']")
```

acl (*permits = True* **,** *explode = False* **,** *resolve = False* **,** *admin = False* **,** *access = None* **,** *allows = None* **,** *sort = 'profile'* **)**
transform the {dataset,general}[Conditional]Access ProfileFrame into an access control list Frame

Parameters :

- **permits** (*bool*) – True: show normal ACL (with the groups identified as `-group-` in the USER_ID column).

- **explode** (*bool*) – True: replace each groups with the users connected to the group (in the USER_ID column). A user ID may occur several times in USER_ID with various ACCESS levels.

- **resolve** (*bool*) – True: show user specific permit, or the highest group permit for each user.

- **admin** (*bool*) – True: add the users that have ability to change the profile or the groups on the ACL (in the

ADMIN_ID column), VIA identifies the group name,
AUTHORITY the RACF privilege involved.

- **access** (*str*) – show entries that are equal to the access level specified, e.g., access='CONTROL'.

- **allows** (*str*) – show entries that are higher or equal to the access level specified, e.g., allows='UPDATE'.

- **sort** (*str*) – sort the resulting output by column: user, access, id, admin, profile.

pyracf.profile_publishers module

class pyracf.profile_publishers. ProfileSelectionFrame

Bases: `object`

Data selection methods to retrieve one profile, or profiles, from a ProfileFrame, using exact match.

These methods typically have a name referring to the singular.

The parameter(s) to these methods are used as a literal search argument, and return entries that fully match the argument(s). Selection criteria have to match the profile exactly, generic patterns are taken as literals.

The number of selection parameters depends on the ProfileFrame, matching the number of index fields in the ProfileFrame. When you specify a parameter as None or '**', the level is ignored in the selection.

The optional parameter `option='LIST'` causes a pandas Series to be returned if there is one matching profile, instead of a ProfileFrame. This is meant for high-performance, looping applications.

group (*group = None* , *option = None*) → ProfileFrame

data frame with 1 record from `.groups` when the group is found, or an empty frame.

Example

```
r.group('SYS1')
```

user (*userid = None* , *option = None*) → ProfileFrame

data frame with 1 record from `.users` when the user ID is found, or an empty frame.

Example

```
r.user('IBMUSER')
```

connect (*group = None* , *userid = None* , *option = None*) → ProfileFrame

> data frame with record(s) from `.connectData` , fitting the parameters exactly, or an empty frame.
>
> Example
>
> ```
> r.connect('SYS1','IBMUSER')
> ```
>
> If one of the parameters is written as `None` , or the second parameter is omitted, all profiles matching the specified parameter are shown, with one index level instead of the 2 index levels that .connectData holds. For example, `r.connect('SYS1')` shows all users connected to SYS1, whereas `r.connect(None, 'IBMUSER')` shows all the groups IBMUSER is member of. Instead of `None` , you may specify `'**'` .
>
> `connect('SYS1')` returns 1 index level with user IDs. `connect(None,'IBMUSER')` or `connect(userid='IBMUSER')` returns 1 index level with group names.
>
> You can find all entries in `.users` that have a group connection to SYSPROG as follows:
>
> ```
> r.users.loc[r.users.USBD_NAME.isin(r.connect('SYSPROG').index)]
> ```
>
> or
>
> ```
> r.users.query("_NAME in @r.connect('SYSPROG').index")
> ```
>
> These forms use the index structure of `.connect` , rather than the data, giving better speed. The 2nd example references the index field `_NAME` rather than the data column `USBD_NAME` .

dataset (*profile = None* , *option = None*) → ProfileFrame

> data frame with 1 record from `.datasets` when a profile is found, fitting the parameters exactly, or an empty frame.
>
> Example
>
> ```
> r.dataset('SYS1.*.**')
> ```
>
> To show all dataset profiles starting with SYS1 use:
>
> ```
> r.datasets.find('SYS1.**')
> ```
>
> To show the dataset profile covering SYS1.PARMLIB use:
>
> ```
> r.datasets.match('SYS1.PARMLIB')
> ```

To find the access control list (acl) of profiles, use the `.acl()` method on any of these selections, e.g.:

```
r.dataset('SYS1.*.**').acl()
```

datasetPermit (*profile = None* **,** *id = None* **,** *access = None* **,** *option = None* **)** → ProfileFrame

data frame with records from `.datasetAccess`, fitting the parameters exactly, or an empty frame

Example

```
r.datasetPermit('SYS1.*.**', None, 'UPDATE')
```

This shows all IDs with update access on the `SYS1.*.**` profile (if this exists). To show entries from all dataset profiles starting with SYS1 use:

```
r.datasetAccess.find('SYS1.**', '**', 'UPDATE')
```

or

```
r.datasets.find('SYS1.**').acl(access='UPDATE')
```

datasetConditionalPermit (*profile = None* **,** *id = None* **,** *access = None* **,** *option = None* **)** → ProfileFrame

data frame with records from `.datasetConditionalAccess`, fitting the parameters exactly, or an empty frame.

Example

```
r.datasetConditionalPermit('SYS1.*.**', None, 'UPDATE')
```

To show entries from all conditional permits for `ID(*)` use:

```
r.datasetConditionalAccess.find('**', '*', '**')
```

general (*resclass = None* **,** *profile = None* **,** *option = None* **)** → ProfileFrame

data frame with profile(s) from `.generals` fitting the parameters exactly, or an empty frame.

Example

```
r.general('FACILITY', 'BPX.**')
```

If one of the parameters is written as `None` or `'**'`, or the second parameter is omitted, all profiles matching the specified parameter are shown:

```
r.general('UNIXPRIV')
```

To show the general resource profile controlling dynamic superuser, use:

```
r.general('FACILITY').match('BPX.SUPERUSER')
```

To show more general resource profiles relevant to z/OS UNIX use:

```
r.generals.find('FACILITY', 'BPX.**')
```

generalPermit (*resclass = None* , *profile = None* , *id = None* , *access = None* , *option = None*) → ProfileFrame

data frame with records from `.generalAccess` , fitting the parameters exactly, or an empty frame.

Example

```
r.generalPermit('UNIXPRIV', None, None, 'UPDATE')
```

This shows all IDs with update access on the any UNIXPRIV profile (if this exists). To show entries from all TCICSTRN profiles starting with CICSP use:

```
r.generalAccess.find('TCICSTRN', 'CICSP*')
```

generalConditionalPermit (*resclass = None* , *profile = None* , *id = None* , *access = None* , *option = None*) → ProfileFrame

data frame with records from `.generalConditionalAccess` fitting the parameters exactly, or an empty frame.

Example

```
r.generalConditionalPermit('FACILITY')
```

To show entries from all conditional permits for `ID(*)` use one of the following:

```
r.generalConditionalPermit('**', '**', '*', '**')
```

```
r.generalConditionalPermit(None, None, '*', None)
```

```
r.generalConditionalAccess.find(None, None, '*', None)
```

```
r.generalConditionalAccess.find(None, None, re.compile('\*'), None)
```

class pyracf.profile_publishers. ProfileAnalysisFrame

Bases: `object`

These properties present a subset of a DataFrame, or the result of DataFrame intersections, to identify points of interest.

The properties do not support parameters, but you can chain a .find() or .skip() method to filter the results.

property **groupsWithoutUsers** *: ProfileFrame*

DataFrame with all groups that have no user IDs connected (empty groups).

property **ownertree** *: GroupStructureTree*

> dict with the user IDs that own groups as key, and a list of their owned groups as values. if a group in this list owns groups, the entry is replaced by a dict.

property **grouptree** *: GroupStructureTree*

> dict starting with SYS1, and a list of groups owned by SYS1 as values. if a group in this list owns groups, the entry is replaced by a dict. because SYS1s superior group is blank/missing, we return the first group that is owned by "".

property **specials** *: ProfileFrame*

> DataFrame (like `.users`) with all users that have the 'special attribute' set. Effectively this is the same as the result from:

```
r.users.loc[r.users['USBD_SPECIAL'] == 'YES']
```

property **operations** *: ProfileFrame*

> DataFrame (like `.users`) with all users that have the 'operations attribute' set.

property **auditors** *: ProfileFrame*

> DataFrame with all users that have the 'auditor attribute' set.

property **revoked** *: ProfileFrame*

> Returns a DataFrame with all revoked users.

property **uacc_read_datasets** *: ProfileFrame*

> DataFrame with all dataset definitions that have a Universal Access of 'READ'

property **uacc_update_datasets** *: ProfileFrame*

> DataFrame with all dataset definitions that have a Universal Access of 'UPDATE'

property **uacc_control_datasets** *: ProfileFrame*

> DataFrame with all dataset definitions that have a Universal Access of 'CONTROL'

property **uacc_alter_datasets** *: ProfileFrame*

> DataFrame with all dataset definitions that have a Universal Access of 'ALTER'

property **orphans** *: tuple*

> IDs on access lists with no matching USER or GROUP entities, in a tuple with 2 RuleFrames

> Legacy code for backward comptibility. This function demonstrates how to access columns in the raw data frames, though definitely not efficiently. FIXED: Temporary frames are used to prevent updating the original _datasetAccess and _generalAccess frames. The functionality is also, and generalized, available in RuleVerifier.

class pyracf.profile_publishers. EnhancedProfileFrame

Bases: `object`

Profile presentation properties that make data easier to report by adding fields to the original ProfileFrame.

property **groupOMVS** *: ProfileFrame*

> add column `GPOMVS_GID_` into `._groupOMVS` with leading zeros stripped from `GPOMVS_GID` .

property **connectData** *: ProfileFrame*

> complete connect group information
>
> Combines fields from USER profiles (0205) and GROUP profiles (0102). The `GPMEM_AUTH` field shows group connect authority, whereas all other field names start with `USCON` . This property should be used for most connect group analysis, instead of `.connects` and `.groupConnect` .

property **userOMVS** *: ProfileFrame*

> add column `USOMVS_UID_` into `._userOMVS` with leading zeros stripped from `USOMVS_UID` .

property **datasets** *: ProfileFrame*

> unspecifiec access columns added to .datasets Frame
>
> Column `IDSTAR_ACCESS` is added by selecting records from `.datasetAccess` referencing `ID(*)` . The higher value of `DSBD_UACC` and `IDSTAR_ACCESS` is stored in `ALL_USER_ACCESS` indicating the access level granted to all RACF defined users, except when restricted by specific access.

property **generals** *: ProfileFrame*

> unspecifiec access columns added to .generals Frame
>
> Column `IDSTAR_ACCESS` is added by selecting records from `.generalAccess` referencing `ID(*)` . The higher value of `GRBD_UACC` and `IDSTAR_ACCESS` is stored in `ALL_USER_ACCESS` indicating the access level granted to all RACF defined users, except when restricted by specific access.

property **CERT** *: ProfileFrame*

> combined DataFrame of `._generalCERT` and `.generals` , copying the `GRBD_UACC` and `GRBD_APPL_DATA` fields to show if the certificate is trusted, and the associated user ID.

property **KEYRING** *: ProfileFrame*

> combined DataFrame of `._generalKEYRING` and `.generals` , copying the `GRBD_APPL_DATA` field to show the associated user ID.

property **SSIGNON** *: ProfileFrame*

combined DataFrame of `._generalSSIGNON` and `.generals` , copying the `GRBD_APPL_DATA` field to show if replay protection is available for the passticket.

class pyracf.profile_publishers. ProfilePublisher
Bases: `ProfileSelectionFrame` , `ProfileAnalysisFrame` , `EnhancedProfileFrame`

straight-forward presentation and easy filtered results of Profile Frames from the RACF object. These are hand-crafted additions to the properties automatically defined from _recordtype_info.

pyracf.racf_functions module

pyracf.racf_functions. generic2regex (*selection* , *lenient = '%&*'*)
Change a RACF generic pattern into regex to match with text strings in pandas cells.

Parameters :

lenient – the characters that are (also) taken to be part of the qualifier. use lenient="" to match with dsnames/ resources

pyracf.racf_functions. accessAllows (*level = None*)
return list of access levels that allow the given access

Example

`RACF.accessAllows('UPDATE')` returns `[,'UPDATE','CONTROL','ALTER','-owner-']`

for use in pandas `.query("ACCESS in @RACF.accessAllows('UPDATE')")`

pyracf.racf_functions. rankedAccess (*args*)
translate access levels into integers, add 10 if permit is for the user ID.

could be used in .apply() but would be called for each row, so very very slow

pyracf.rule_verify module

class pyracf.rule_verify. RuleFrame (*data = None* , *index : Axes | None = None* , *columns : Axes | None = None* , *dtype : Dtype | None = None* , *copy : bool | None = None*)
Bases: `DataFrame` , `FrameFilter`

Output of a verify() action

find (*selection* , ** *kwds*)

    Search rule results using GENERIC pattern on the data fields. selection can be one or more values, corresponding to data columns of the df.

    alternatively specify the field names via an alias keyword (resclass, profile, field, actual, found, expect, fit, value or id):

```
r.rules.load().verify().find(field='OWN*')
```

    specify selection as regex using re.compile:

```
r.rules.load().verify().find( field=re.compile('(OWNER|DFLTGRP)' )
```

skip (*selection* , ** *kwds*)

    Exclude rule results using GENERIC pattern on the data fields. selection can be one or more values, corresponding to data columns of the df

    alternatively specify the field names via an alias keyword (resclass, profile, field, actual, found, expect, fit, value or id):

```
r.rules.load().verify().skip(actual='SYS1')
```

class pyracf.rule_verify. RuleVerifier (*RACFobject*)

    Bases: `object`

verify fields in profiles against expected values, issues are returned in a df.

rules can be passed as a dict of [tuples or lists], and a dict with domains, via parameter, or as function result from external module. created from RACF object with the .rules property.

load (*rules = None* , *domains = None* , *module = None* , *reset = False* , *defaultmodule = ':profile_field_rules'*)

    load rules + domains from yaml str, structure or from packaged module

    Parameters :

- **rules** (*dict* , *str*) – dict of tuples or lists with test specifications, or yaml str field that expands into a dict of lists

- **domains** (*dict* , *str*) – one or more domain in a dict(name=[entries]), or in a yaml string

- **module** (*str*) – name of module that contains functions rules() and domains()

- **defaultmodule** (*str*) – module name to be used if all parameters are omitted

- **reset** (*bool*) – clear rules, domains and module in RuleVerifier object, before loading new values

Returns :

the updated object

Return type :

RuleVerifier

Example:

```
r.rules.load(rules = {'test libraries':
    (['DSBD'],
     {'id': '101',
      'rule': 'Integrity of test libraries',
      'profile': 'TEST*.**',
      'test': [{'field':'UACC', 'value':['NONE','READ']},
              {'field':'WARNING', 'value':'NO'},
              {'field':'NOTIFY_ID', 'fit':'DELETE'}],
     }
    )
              }
       ).verify()
```

add_domains (*domains = None*)
    Add domains to the end of the domain list, from a dict or a yaml string value.

Parameters :

domains (*dict* , *str*) – one or more domains in a dict(name=[entries]), or in a yaml string

Returns :

The updated object

Return type :

RuleVerifier

Example:

```
v = r.rules.load()

v.add_domains({'PROD_GROUPS': ['PRODA','PRODB','PRODCICS'],
               'TEST_GROUPS': ['TEST1','TEST2']})

v.add_domains({'SYS1': r.connect('SYS1').index})
```

get_domains (*domains = None*)

Get domain definitions as a dict, or one entry as a list.

Parameters :

str (*domains*) – name of domain entry to return as list,
or None to return all

Returns :

dict or list

Example:

```
v.get_domains() # all domains as a dict

v.get_domains('PROD_GROUPS') # one domain as a list
```

verify (*rules = None* , *domains = None* , *module = None* , *reset = False* , *id = True* , *verbose =*
False , *syntax_check = None* , *optimize = 'rows cols'*) → RuleFrame

verify fields in profiles against the expected value, issues are returned in a df

Parameters :

- **id** (*bool*) – False: suppress ID column from the result
  frame. The values in this column are taken from the id
  property in rules

- **syntax_check** (*bool*) – deprecated

- **verbose** (*bool*) – True: print progress messages

- **optimize** (*str*) – cols to improve join speed, rows to
  use pre-selection

Returns :

Result object (RuleFrame)

Example:

```
r.rules.load().verify()
```

syntax_check (*confirm = True*) → RuleFrame

parse rules and domains, check for consistency and unknown directives, normalize field names

specify confirm=False to suppress the message when all is OK

Parameters :

confirm (*bool*) – False if the success message should be suppressed, so in automated testing the result frame has .empty

Returns :

syntax messages (RuleFrame)

Example:

```
r.rules.load().syntax_check()

if r.rules.load().syntax_check(confirm=False).empty:
    print('No syntax errors in default policy')
```

pyracf.utils module

pyracf.utils. deprecated (*func* , *oldname*)

Wrapper routine to add (deprecated) alias name to new routine (func), supports methods and properties. Inspired by functools.partial()

pyracf.utils. listMe (*item*)

make list in parameters optional when there is only 1 item, similar to the * unpacking feature in assignments. as a result you can just: for options in listMe(optioORoptions)

pyracf.utils. readableList (*iter*)

print entries from a dict index into a readable list, e.g., a, b or c

pyracf.utils. simpleListed (*item*)

> print a string or a list of strings with just commas between values

pyracf.utils. nameInColumns (*df* , *name* , *columns = []* , *prefix = None* , *returnAll = False*)

> find prefixed column name in a Frame, return whole name, or all names if requested

> Parameters :

> > · **df** – Frame to find column names, or None

> > · **name** (*str , list*) – name to search for, with prefix or without, or list of names

> > · **columns** (*list*) – opt. ignore df parameter, caller has already extracted column names

> > · **prefix** (*str , list*) – opt. verify that column name has the given prefix(es)

> > · **returnAll** (*bool*) – always return all matches in a list

> Returns :

> fully prefixed column name, or list of column names

pyracf.xls_writers module

class pyracf.xls_writers. XlsWriter

> Bases: `object`

> accessMatrix2xls (*fileName = 'irrdbu00.xlsx'*)

> > create excel file with sheets for each class, lines for each profile and columns for each ID in the access control list. runs as method on the RACF object, table('DSACC') or table('GRACC')

> xls (*** keywords*)

Module contents

class pyracf. RACF (*irrdbu00 = None* , *pickles = None* , *auto_pickles = False* , *prefix = ''*)

> Bases: `ProfilePublisher` , `XlsWriter`

STATE_BAD = *-1*
STATE_INIT = *0*
STATE_PARSING = *1*
STATE_CORRELATING = *2*
STATE_CORRELATED = *3*
STATE_READY = *4*
property **status**
parse_fancycli (*recordtypes = None* , *save_pickles = False* , *prefix = ''*)
parse (*recordtypes = None*)
parse_t (*thingswewant = None*)
parsed (*rname*)
> how many records with this name (type) were parsed

table (*rname = None*) → ProfileFrame
> return table with this name (type)

save_pickle (*df = ''* , *dfname = ''* , *path = ''* , *prefix = ''*)
save_pickles (*path = '/tmp'* , *prefix = ''*)
load_pickles (*path = '/tmp'* , *prefix = ''*)
property **rules** *: RuleVerifier*
> create a RuleVerifier instance

getdatasetrisk (*profile = ''*)
> This will produce a dict as follows:

property **ALIAS** *: ProfileFrame*
> general resource ALIAS group (05B0).

property **CDTINFO** *: ProfileFrame*
> general resource CDTINFO data (05C0).

property **CERTname** *: ProfileFrame*
> general resource certificate information (1560).

property **CERTreferences** *: ProfileFrame*
> Certificate References (0561).

property **CFDEF** *: ProfileFrame*
> general resource CFDEF data (05E0).

property **DLFDATA** *: ProfileFrame*
> General Resources DLF Data (0520).

property **DLFDATAjobnames** *: ProfileFrame*
> General Resources DLF Job Names (0521).

property **DistributedIdFilter** *: ProfileFrame*

Filter Data (0508).

property **DistributedIdMapping** *: ProfileFrame*

General Resource Distributed Identity Mapping Data (0509).

property **EIM** *: ProfileFrame*

general resource EIM segment (05A0).

property **ICSF** *: ProfileFrame*

general resource ICSF (05G0).

property **ICSFsymexportCertificateIdentifier** *: ProfileFrame*

general resource ICSF certificate identifier (05G2).

property **ICSFsymexportKeylabel** *: ProfileFrame*

general resource ICSF key label (05G1).

property **ICTX** *: ProfileFrame*

general resource ICTX segment (05D0).

property **IDTFPARMS** *: ProfileFrame*

Identity Token data (05K0).

property **JES** *: ProfileFrame*

JES data (05L0).

property **KERB** *: ProfileFrame*

general resource KERB segment (0580).

property **MFA** *: ProfileFrame*

Multifactor factor definition data (05H0)

property **MFPOLICY** *: ProfileFrame*

Multifactor Policy Definition data (05I0).

property **MFPOLICYfactors** *: ProfileFrame*

user Multifactor authentication policy factors (05I1).

property **PROXY** *: ProfileFrame*

general resource PROXY (0590).

property **SESSION** *: ProfileFrame*

General Resources Session Data (0510).

property **SESSIONentities** *: ProfileFrame*

General Resources Session Entities (0511).

property **SIGVER** *: ProfileFrame*

general resource SIGVER data (05F0).

property **STDATA** *: ProfileFrame*

Record type (0540).

property **SVFMR** *: ProfileFrame*

Record type (0550).

property **TME** *: ProfileFrame*

general resource TME data (0570).

property **TMEchild** *: ProfileFrame*

general resource TME child (0571).

property **TMEgroup** *: ProfileFrame*

general resource TME group (0573).

property **TMEresource** *: ProfileFrame*

general resource TME resource (0572).

property **TMErole** *: ProfileFrame*

general resource TME role (0574).

property **connectData** *: ProfileFrame*

User Connect Data (0205).

property **connects** *: ProfileFrame*

Group Members (0102).

property **datasetAccess** *: ProfileFrame*

Data Set Access (0404).

property **datasetCSDATA** *: ProfileFrame*

Data Set CSDATA custom fields (0431).

property **datasetCategories** *: ProfileFrame*

Data Set Categories (0401).

property **datasetConditionalAccess** *: ProfileFrame*

Data Set Conditional Access (0402).

property **datasetDFP** *: ProfileFrame*

Data Set DFP Data (0410).

property **datasetMember** *: ProfileFrame*

Data Set Member Data (0406).

property **datasetTME** *: ProfileFrame*

Data Set TME Data (0421).

property **datasetUSRDATA** *: ProfileFrame*

Data Set Installation Data (0405).

property **datasetVolumes** *: ProfileFrame*

Data Set Volumes (0403).

property **datasets** *: ProfileFrame*

Data Set Basic Data (0400).

property **generalAccess** *: ProfileFrame*

General Resource Access (0505).

property **generalCSDATA** *: ProfileFrame*

General Resources CSDA custom fields (05J1).

property **generalCategories** *: ProfileFrame*

General Resources Categories (0502).

property **generalConditionalAccess** *: ProfileFrame*

General Resources Conditional Access (0507).

property **generalMembers** *: ProfileFrame*

General Resource Members (0503).

property **generalTAPEvolume** *: ProfileFrame*

General Resource Tape Volume Data (0501).

property **generalTAPEvolumes** *: ProfileFrame*

General Resources Volumes (0504).

property **generalUSRDATA** *: ProfileFrame*

General Resource Installation Data (0506).

property **generals** *: ProfileFrame*

General Resource Basic Data (0500).

property **groupCSDATA** *: ProfileFrame*

Group CSDATA custom fields (0151).

property **groupConnect** *: ProfileFrame*

User Group Connections (0203).

property **groupDFP** *: ProfileFrame*

Group DFP Data (0110).

property **groupOVM** *: ProfileFrame*

Group OVM Data (0130).

property **groupTME** *: ProfileFrame*
Group TME Data (0141).

property **groupUSRDATA** *: ProfileFrame*
Group Installation Data (0103).

property **groups** *: ProfileFrame*
Group Basic Data (0100).

property **subgroups** *: ProfileFrame*
Group Subgroups (0101).

property **userAssociationMapping** *: ProfileFrame*
User Associated Mappings (0208).

property **userCERTname** *: ProfileFrame*
user certificate name (0207).

property **userCICS** *: ProfileFrame*
User CICS Data (0230).

property **userCICSoperatorClasses** *: ProfileFrame*
User CICS Operator Class (0231).

property **userCICSrslKeys** *: ProfileFrame*
User CICS RSL keys (0232).

property **userCICStslKeys** *: ProfileFrame*
User CICS TSL keys (0233).

property **userCSDATA** *: ProfileFrame*
user CSDATA custom fields (02G1).

property **userCategories** *: ProfileFrame*
User Categories (0201).

property **userClasses** *: ProfileFrame*
User Classes (0202).

property **userDCE** *: ProfileFrame*
user DCE data (0290).

property **userDFP** *: ProfileFrame*
User DFP data (0210).

property **userDistributedIdMapping** *: ProfileFrame*

User Associated Distributed Mappings (0209).

property **userEIM** *: ProfileFrame*
    user EIM segment (02F0).

property **userKERB** *: ProfileFrame*
    User KERB segment (02D0).

property **userLANGUAGE** *: ProfileFrame*
    User Language Data (0240).

property **userLNOTES** *: ProfileFrame*
    LNOTES data (02B0).

property **userMFAfactor** *: ProfileFrame*
    user Multifactor authentication data (020A).

property **userMFAfactorTags** *: ProfileFrame*
    user Multifactor authentication factor configuration data (1210).

property **userMFApolicies** *: ProfileFrame*
    user Multi-factor authentication policies (020B)

property **userNDS** *: ProfileFrame*
    NDS data (02C0).

property **userNETVIEW** *: ProfileFrame*
    user NETVIEW segment (0280).

property **userNETVIEWdomains** *: ProfileFrame*
    user DOMAINS (0282).

property **userNETVIEWopclass** *: ProfileFrame*
    user OPCLASS (0281).

property **userOPERPARM** *: ProfileFrame*
    User OPERPARM Data (0250).

property **userOPERPARMscope** *: ProfileFrame*
    User OPERPARM Scope (0251).

property **userOVM** *: ProfileFrame*
    user OVM data (02A0).

property **userPROXY** *: ProfileFrame*
    user PROXY (02E0).

property **userRRSFDATA** *: ProfileFrame*

RRSF data (0206).

property **userTSO** *: ProfileFrame*

User TSO Data (0220).

property **userUSRDATA** *: ProfileFrame*

User Installation Data (0204).

property **userWORKATTR** *: ProfileFrame*

User WORKATTR Data (0260).

property **users** *: ProfileFrame*

User Basic Data (0200).

class pyracf. IRRDBU (*irrdbu00 = None* , *pickles = None* , *auto_pickles = False* , *prefix = ''*)

Bases: `RACF`

Indices and tables

- Index

- Module Index

- Search Page