IDIS Core Documentation

Release 1.0.3

Sjoerd Kerkstra

Oct 18, 2022

CONTENTS:

IDIS core de-identifies DICOM datasets. It does this by removing or replacing DICOM elements when needed. All DICOM processing is based on pydicom. It processes in accordance to the DICOM deidentification profile and options.

It works like this:

```
import pydicom
from idiscore.defaults import create_default_core

core = create_default_core()      # create an idiscore instance

ds = pydicom.dcmread("my_file.dcm")  # load a DICOM dataset
ds = core.deidentify(ds)          # remove patient information
ds.save_as("deidentified.dcm")    # save to disk
```

See *Getting started* to start using idiscore

CHAPTER

ONE

GOALS

- Deidentify DICOM datasets in conformance to the DICOM standard
- Configuration is an extra, not a requirement. With minimal configuration IDIS core should deidentify a dataset 'well' This means IDIS core will include opinions on deidentification
- Python all the way. No custom configuration languages, no installers, just scripts and pip. IDIS core assumes you can write python and leverages class inheritance, docstrings, pytest, variable annotations. This keeps things clean, testable and unambiguous

CHAPTER

TWO

NON-GOALS

- No deidentification pipeline. IDIS core deidentifies DICOM datasets. It does not want to know where this dataset comes from or where it is going to. It does not offer any installable or server to send files to. It could be used to create such a server, but this is out of this project's scope
- Reading and Writing DICOM files. Internally IDIS core only works with pydicom datasets. Reading and writing of DICOM datasets is to pydicom

CHAPTER

THREE

ALTERNATIVES

Alternative methods of de-identification

CTP

MIRC CTP is a widely used, extensive, java-based framework for deidentification and data aggragation. It has many plugins and can be configured using several scripting languages. All in all it is a very good choice for many people. For me as a programmer developing mostly python-based software, I struggled with certain aspects however:

- It is difficult to integrate into a test suite properly. This is first of all because it is file-based, requiring an actual file on disk for each type of DICOM you might want to verify the deidentification of. Second, because the pipeline is configured with several different file-based custom scripts it is difficult to set up the correct context for tests.
- I found it tricky to integrate into my python-based infrastructure. Again, because the pipeline is java-based and file-based there is no easy way to access the state of files in the pipeline. Is a file done? Has something gone wrong? Getting this information would require either checking all possible output, stage and quarantine folders. I was really missing exceptions I could catch.
- Because it is an installable pipeline, I found it difficult to integrate into smaller, non-server based applications like a command line tool that locally deidentifies some data for a user.

deid

pydicom deid is a pydicom based best-effort anonymizer for medical image data. It is part of the pydicom family. It has extensive and friendly documentation and get several concepts right. Reasons for not expanding on this library and instead starting a new one:

- There seems to have been little development since the libraries start in 2017
- Seems to be quite file-based in places, often requiring input and output folders for initializing objects
- No test coverage monitoring, uses unittest for testing which is hard to maintain and expand on
- Uses custom scripting language for configuring the anonymization. This is useful for non-coding end-users, but adds a layer of indirectness to automated testing.

3.1 IDIS Core

Deidentification of DICOM images using Attribute Confidentiality Options

- Free software: GPLv3 License
- Documentation: https://idiscore.readthedocs.io.

3.1.1 Features

- Pure-python de-identification using pydicom
- De-identification is verified by test suite
- Useful even without configuration - offers reasonable de-identification out of the box.
- Uses standard DICOM Confidentiality options to define de-identification that is to be performed
- Focus on de-identification, pydicom dataset in -> pydicom dataset out. No pipeline management, No special input and output handling.

3.1.2 Credits

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

3.2 Getting started

3.2.1 Installation

```
$ pip install idiscore
```

For more details see installation

3.2.2 How to run idiscore

Idiscore is meant to be used within a python script:

```
import pydicom
from idiscore.defaults import create_default_core

core = create_default_core()      # create an idiscore instance

ds = pydicom.dcmread("my_file.dcm")  # load a DICOM dataset
```

(continues on next page)

(continued from previous page)

```
ds = core.deidentify(ds)          # remove patient information
ds.save_as("deidentified.dcm")    # save to disk
```

3.2.3 Choosing a deidentification profile

Deidentification is based on the DICOM standard deidentification profile and one or more DICOM Confidentiality options. The minimal example above uses the idiscore default profile which uses some of these options (defined as 'rule sets').

To select DICOM confidentiality options yourself, initialise a core instance like this:

```
from idiscore.core import Core, Profile
from idiscore.defaults import get_dicom_rule_sets

sets = get_dicom_rule_sets()  # Contains official DICOM deidentification rules
profile = Profile(            # Choose which rule sets to use
    rule_sets=[sets.basic_profile,
               sets.retain_modified_dates,
               sets.retain_device_id]
)
core = Core(profile)          # Create an deidentification core
```

The rule sets in idiscore implement the rules in DICOM PS3.15 table E.1-1.

3.2.4 Safe Private and PII location list

Safe private and PII location lists are often needed for more advanced deidentification. They address two special types of data:

Private DICOM tags
: These are non-standard tags that can be written into a DICOM dataset by any manufacturer. A list of private tags considered safe can be passed to an idiscore instance. Without this list idiscore will remove all private tags

PixelData
: In certain types of DICOM datasets, Personally Identifiable information (PII) is burnt into the image itself. This is often the case for ultrasound images for example. To handle this a list of known PII locations can be passed to an idiscore instance. Without this list, datasets with burnt-in information will be rejected

Here is an example of passing both lists to an idiscore instance:

```
from idiscore.defaults import create_default_core
from idiscore.image_processing import PIILocation, PIILocationList, SquareArea
from idiscore.private_processing import SafePrivateBlock, SafePrivateDefinition

safe_private = SafePrivateDefinition(
    blocks=[
        SafePrivateBlock(
            tags=["0023[SIEMENS MED SP DXMG WH AWS 1]10",
                  "0023[SIEMENS MED SP DXMG WH AWS 1]11",
                  "00b1[TestCreator]01",
                  "00b1[TestCreator]02"],
            criterion=lambda x: x.Modality == "CT",
```

(continues on next page)

(continued from previous page)

```
            comment='Some test tags, only valid for CT datasets'),
        SafePrivateBlock(
            tags=["00b1[othercreator]11", "00b1[othercreator]12"],
            comment='Some more test tags, without a criterion')])

location_list = PIILocationList(
    [PIILocation(
        areas=[SquareArea(5, 10, 4, 12),
               SquareArea(0, 0, 20, 3)],
        criterion=lambda x: x.Rows == 265 and x.Columns == 512
     ),
     PIILocation(
        areas=[SquareArea(0, 200, 4, 12)],
        criterion=lambda x: x.Rows == 265 and x.Columns == 712
     )]
)

core = create_default_core(safe_private_definition=safe_private,
                           location_list=location_list)
```

Tip: When passing a safe private definition, make sure the rule set *Retain Safe Private* is included in your profile

For more information on how idiscore works, see *Advanced*.

3.3 Advanced

More in-depth discussion of on certain issues. Intended for people interested in customising what idiscore does

3.3.1 How idiscore deidentifies a dataset

Getting a sense of what the method `idiscore.core.Core.deidentify()` actually does. Starting at the very specific.

- A dataset is fed into `idiscore.core.Core.deidentify()` on a `default idiscore instance`. What will happen?
- Suppose that the dataset contains the DICOM element *0010, 0010 (PatientName) - Jane Smith*
- An *idiscore.operators.Operator()* is applied to this element. In the default case this is *idiscore.operators.Empty()*. This will keep the element, but remove its value.
- the *Empty* operator was applied because the default profile has the *Rule* *0010, 0010 (PatientName) - Empty*

Overview

- `idiscore.core.Core.deidentify()` deidentifies a dataset in 4 steps:
  1. `idiscore.core.Core.apply_bouncers()` Can reject a dataset if it is considered too hard to deidentify.
  2. `idiscore.core.Core.apply_pixel_processing()` Removes part of the image data if required. If image data is unknown or something else goes wrong the dataset is rejected
  3. `idiscore.core.Core.apply_rules()` Process all DICOM elements. Remove, replace, keep, according to the profile that was set. See for example all rules for the idiscore default profile. This step is the most involved of the steps listed here. It will be
  4. Insert any new elements into the dataset. *idiscore.insertions.get_deidentification_method()* for example generates an element that indicates what method was used for deidentification

3.3.2 How to modify and extend processing

Custom profile

```
"""You can set your own rules for specific DICOM tags. Be aware that this might

mean the deidentification is no longer DICOM-complient
"""

import pydicom

from idiscore.core import Core, Profile
from idiscore.defaults import get_dicom_rule_sets
from idiscore.identifiers import RepeatingGroup, SingleTag
from idiscore.operators import Hash, Remove
from idiscore.rules import Rule, RuleSet

# Custom rules that will hash the patient name and remove all curve data
my_ruleset = RuleSet(
    rules=[
        Rule(SingleTag("PatientName"), Hash()),
        Rule(RepeatingGroup("50xx,xxxx"), Remove()),
    ],
    name="My Custom RuleSet",
)

sets = get_dicom_rule_sets()  # Contains official DICOM deidentification rules
profile = Profile(  # add custom rules to basic profile
    rule_sets=[sets.basic_profile, my_ruleset]
)
core = Core(profile)  # Create an deidentification core

# read a DICOM dataset from file and write to another
core.deidentify(pydicom.dcmread("my_file.dcm")).save_as("deidentified.dcm")
```

Each *Rule* above consists of two parts: an *Identifier* which designates what this rule applies to, and an *Operator* which defines what the rule does

Custom processing

If the existing *Operators* in *idiscore.operators* are not enough, you can define your own by extending *idiscore.operators.Operator()*. If these operators could be useful for other users as well, please consider creating a pull request (see *Contributing*)

3.4 Concepts

Things in idiscore that are not necessarily code but require more explanation nonetheless

3.4.1 Glossary

Terms used throughout this documentation

IDIS core
: Library that implements basic deidentification. Requires configuration before it can actually be used or deployed. Implements each of the standard DICOM confidentiality options.

DICOM deidentification option
: DICOM Confidentiality options are a part of the DICOM standard which helps describe to which extent data is deidentified. In addition to a compulsory Basic profile there are 10 modifier options which either remove additional data, such as 'Clean Pixel Data' or which remove less data, such as 'Retain Patient Characteristics'.

IDIS core configuration
: All information needed by IDIS core to actually deidentify a DICOM dataset. Safe private tag definitions, the Confidentiality options to use, Pixel data definitions, and any custom additional options.

IDIS core instance
: A specific version of the IDIS core library combined with a specific configuration. This can be deployed and used as is. This is the object that can be validated and tested against a collection of DICOM examples.

DICOM example
: An annotated DICOM dataset. The annotations indicate for one or more DICOM tags whether the tag contains personal information or not. A DICOM example can be used to verify deidentification

IDIS verify
: A library that can run one or more DICOM examples through an IDIS core instance and test whether deidentification is correct according to each example. Produces a Data Certificate Potentially also determines which

Data certificate
: A list of DICOM examples which have been successfully passed through a IDIS Core instance IDIS Verify. For these examples the Core Instance is 'certified' to work properly. The data certificate can also be used to determine whether new data can be processed or not

DICOM example library
: A collection of DICOM examples

DICOM example tool
: CLI tool that makes it easy to collect, anonymize and annotate DICOM examples

PII
: Personally Identifiable Information. Information in a DICOM dataset that can be used to trace back the dataset to a single person. Deidentification attempts to remove all such information

3.5 modules

All modules in idiscore

3.5.1 idiscore.annotation module

3.5.2 idiscore.bouncers module

class idiscore.bouncers.**Bouncer**

Bases: object

Inspects a dataset and either rejects it or lets it through

description = 'Bouncer'

inspect(*dataset: Dataset*)

Check given dataset, raise exception if it should be rejected

Parameters
dataset (*Dataset*) – The DICOM dataset to inspect

Return type
None

Raises
BouncerException – When this dataset cannot be deidentified for any reason

exception idiscore.bouncers.**BouncerException**

Bases: *IDISCoreError*

class idiscore.bouncers.**RejectEncapsulatedImageStorage**

Bases: *Bouncer*

description = 'Reject encapsulated PDF and CDA'

inspect(*dataset: Dataset*)

Check given dataset, raise exception if it should be rejected

Parameters
dataset (*Dataset*) – The DICOM dataset to inspect

Return type
None

Raises
BouncerException – When this dataset cannot be deidentified for any reason

class idiscore.bouncers.**RejectKOGSPS**

Bases: *Bouncer*

description = 'Reject PresentationStorage and KeyObjectSelectionDocument'

inspect(*dataset: Dataset*)

Rejects three types of DICOM objects: 1.2.840.10008.5.1.4.1.1.11.1 - GrayscaleSoftcopyPresentationStateStorage 1.2.840.10008.5.1.4.1.1.88.59 - KeyObjectSelectionDocumentStorage 1.2.840.10008.5.1.4.1.1.11.2 - ColorSoftcopyPresentationStateStorage These often contain ids and physician names in their SeriesDescription. See ticket #8465

Raises
BouncerException – When the dataset is one of these types

class idiscore.bouncers.**RejectNonStandardDicom**

Bases: *Bouncer*

description = 'Reject non-standard DICOM types by SOPClassUID'

inspect(*dataset: Dataset*)

Reject all DICOM that is not one of the standard SOPClass types.

All standard types are listed in DICOM PS3.4 section 5B: http://dicom.nema.org/dicom/2013/output/chtml/part04/sect_B.5.html

idiscore.bouncers.**handle_required_tag_not_found**(*func*)

Decorator for handling missing dataset keys, together with RequiredDataset()

Reduces duplicated code in most Bouncer.inspect() definitions

3.5.3 idiscore.core module

3.5.4 idiscore.dataset module

Additions to the pydicom Dataset object

class idiscore.dataset.**RequiredDataset**(**args: Union[Dataset, MutableMapping[BaseTag, Union[DataElement, RawDataElement]]], **kwargs: Any*)

Bases: Dataset

A pydicom Dataset,that raises distinctive errors when accessing missing keys

Made this to specifically handle missing keys on a dataset. By default a Dataset instance raises KeyError and AttributeError. These are too general to safely catch over larger pieces of code. Putting try except blocks around each individual dict key access is ugly and annoying.

Raises
RequiredTagNotFound – When a requested key is not found in this dataset. Either through attribute access, like dataset.PatientID or through dict access like dataset['PatientID']

Notes

Init like this:

```
>>> ds = Dataset()
>>> rds = RequiredDataset(ds)
```

Now you can handle missing keys cleanly without accidentally catching other KeyErrors:

```
>>> try:
>>>     important_dataset_check(rds)
>>> except RequiredTagNotFound:
>>>     print('check failed due to missing information')
```

exception idiscore.dataset.**RequiredTagNotFound**

Bases: *IDISCoreError*

3.5.5 idiscore.defaults module

3.5.6 idiscore.delta module

class idiscore.delta.**Delta**(*tag: BaseTag*, *before*, *after*)

Bases: `object`

A change in a DICOM element value after deidentification

full_description() → str

Full human-readable description of the change that happened

has_changed() → bool

Has changed or has been removed after deidentification

property status: str

property tag_name: str

class idiscore.delta.**DeltaStatusCodes**

Bases: `object`

How has a DICOM element changed?

ALL = {'CHANGED', 'CREATED', 'EMPTIED', 'REMOVED', 'UNCHANGED'}

CHANGED = 'CHANGED'

CREATED = 'CREATED'

EMPTIED = 'EMPTIED'

REMOVED = 'REMOVED'

UNCHANGED = 'UNCHANGED'

3.5.7 idiscore.exceptions module

exception idiscore.exceptions.**AnnotationValidationFailedError**

Bases: *IDISCoreError*

exception idiscore.exceptions.**IDISCoreError**

Bases: `Exception`

Base for all exceptions in IDIS core

exception idiscore.exceptions.**SafePrivateError**

Bases: *IDISCoreError*

3.5.8 idiscore.identifiers module

Ways to designate a DICOM tag or a group of dicom tags

class idiscore.identifiers.**PrivateBlockTagIdentifier**(*tag: str*)

Bases: *TagIdentifier*

A private DICOM tag with a private creator. Like '0013,[MyCompany]01'

In this example [MyCompany] refers whatever block was reserved by private creator identifier 'MyCompany'

For more info on private blocks, see DICOM standard part 5, section 7.8.1 ('Private Data Elements')

```
BLOCK_TAG_REGEX = re.compile('(?P<group>[0-9A-F]{4}),?\\s?\\[(?P<private_creator>.
*)\\](?P<element>[0-9,A-F]*)',
re.IGNORECASE)
```

as_python() → str

For special export. Python code that recreates this instance

classmethod init_explicit(*group: int*, *private_creator: str*, *element: int*)

Create with explicit parameters. This cannot be the main init because TagIdentifier classes need to be instantiable from a single string and uphold cls(cls.tag)=cls

Parameters

- **group** (*int*) – DICOM group, between 0x0000 and 0xFFFF
- **private_creator** (*str*) – Name of the private creator for this tag
- **element** (*int*) – The two final bytes of the element. Between 0x00 and 0xFF

key() → str

For sane sorting, make sure this matches the key format of other identifiers

matches(*element: DataElement*) → bool

True if private element has been created by private creator and the rest of the group and element match up

name() → str

Human readable name for this tag

number_of_matchable_tags() → int

How many tags could this identifier match?

classmethod parse_tag(*tag: str*) → Tuple[int, str, int]

Parses 'xxxx,[creator]yy' into xxxx, creator and yy components. xxxx and yy are interpreted as hexadecimals

Parameters

tag (*str*) – Format: 'xxxx,[creator]yy' where xxxx and yy are hexadecimals. Case insensitive.

Returns

xxxx: int, creator:str and yy:int from tag string 'xxxx,[creator]yy' where xxxx and yy are read as hexadecimals from string

Return type

Tuple[int, str, int]

Raises

ValueError: – When input cannot be parsed

property tag: str

static to_tag(*group: int*, *private_creator: str*, *element: int*) → str

Tag string like '1301,[creator]01' from individual elements

Parameters

- **group** (*int*) – DICOM group, between 0x0000 and 0xFFFF
- **private_creator** (*str*) – Name of the private creator for this tag
- **element** (*int*) – The two final bytes of the element. Between 0x00 and 0xFF

class idiscore.identifiers.**PrivateTags**

Bases: *TagIdentifier*

Matches any private DICOM tag. A private tag has an uneven group number

static as_python() → str

For special export. Python code that recreates this instance

key() → str

String used in comparison operators

Also. A key should contain all information needed to recreate an instance. if 'tag' is a TagIdentifier instance, the following should hold:

```
>>> tag(tag.key()) == tag
```

matches(*element: DataElement*) → bool

The given element matches this identifier

name() → str

Human-readable name for this tag

number_of_matchable_tags() → int

The number of distinct tags that this identifier could match

Used to determine order of matching (specific -> general)

class idiscore.identifiers.**RepeatingGroup**(*tag: Union[str,* RepeatingTag*]*)

Bases: *TagIdentifier*

A DICOM tag where not all elements are filled. Like (50xx,xxxx)

as_python() → str

For special export. Python code that recreates this instance

key() → str

For sane sorting, make sure this matches the key format of other identifiers

matches(*element: DataElement*) → bool

True if the tag values match this repeater in all places without an 'x'

name() → str

Human readable name for this tag

number_of_matchable_tags() → int

The number of distinct tags that this identifier could match

Used to determine order of matching (specific -> general)

class idiscore.identifiers.**RepeatingTag**(*tag: str*)

Bases: object

Dicom tag with x's in it to denote wildcards, like (50xx,xxxx) for curve data

See http://dicom.nema.org/medical/dicom/current/output/chtml/part05/sect_7.6.html

Raises
ValueError – on init if tag cannot be parsed as a DICOM repeater group

Notes

I would prefer to take any pydicom way of working with repeater tags, but the current version of pydicom (2.0) only offers limited lookup support as far as I can see

as_mask() → int

Byte mask that can remove the byte positions that have value 'x'

RepeatingTag('0010,xx10').as_mask() -> 0xffff00ff RepeatingTag('50xx,xxxx').as_mask() -> 0xff000000

name() → str

Human-readable name for this repeater tag, from pydicom lists

number_of_wildcard_positions() → int

Number of x's in this wildcard

static parse_tag_string(*tag: str*) → str

Cleans tag string and outputs it in standard format. Raises ValueError if tag is not of the correct format like (0010,10xx).

Returns
standard format, 8 character hex string with 'x' for wildcard bytes. like 0010xx10 or 75f300xx

Return type
str

static_component() → int

The int value of all bytes of this tag that are not 'x' RepeatingTag('0010,xx10').static_component() -> 0x00100010 RepeatingTag('50xx,xxxx').static_component() -> 0x50000000

class idiscore.identifiers.**SingleTag**(*tag: Union[BaseTag, str, Tuple[int, int]]*)

Bases: *TagIdentifier*

Matches a single DICOM tag like (0010,0010) or 'PatientName'

as_python() → str

For special export. Python code that recreates this instance

key() → str

Return a valid Tag() string argument

matches(*element: DataElement*) → bool

The given element matches this identifier

name() → str

Human-readable name for this tag

number_of_matchable_tags() → int

The number of distinct tags that this identifier could match

Used to determine order of matching (specific -> general)

class idiscore.identifiers.**TagIdentifier**

Bases: object

Identifies a single DICOM tag or repeating group like (50xx,xxx)

Using just DICOM tags is too limited for defining deidentification. We want to be able to represent for example:

- all curves (50xx,xxxx)
- a private tag with private creator group (01[PrivateCreatorName],0010)

as_python() → str

For special export. Python code that recreates this instance

key() → str

String used in comparison operators

Also. A key should contain all information needed to recreate an instance. if 'tag' is a TagIdentifier instance, the following should hold:

```
>>> tag(tag.key()) == tag
```

matches(*element: DataElement*) → bool

The given element matches this identifier

name() → str

Human-readable name for this tag

number_of_matchable_tags() → int

The number of distinct tags that this identifier could match

Used to determine order of matching (specific -> general)

idiscore.identifiers.**clean_tag_string**(*x*)

Remove common clutter from pydicom Tag.__str__() output

idiscore.identifiers.**get_keyword**(*tag*)

Human-readable keyword for known dicom tags, or 'Unknown'

3.5.9 idiscore.imageprocessing module

Classes and methods for working with image part of a DICOM dataset

exception idiscore.image_processing.**CriterionException**

Bases: *IDISCoreError*

class idiscore.image_processing.**PIILocation**(*areas: List[SquareArea]*, *criterion: Optional[Callable[[Dataset], bool]] = None*)

Bases: object

One or more areas in a DICOM image slice that might contain Personally Identifiable Information (PPI)

Notes

A PIILocation is 2D. Cleaning will be done on each slice individually.

Responsibilities:

- Holds location information. Does not alter PixelData itself
- Determine whether it applies to a given Dataset

exists_in(*dataset: Dataset*) → bool

True if the given PII location exists in the given dataset

Raises
CriterionException – If for some reason no True or False response can be given for this dataset

class idiscore.image_processing.**PIILocationList**(*locations: Optional[List[*PIILocation*]] = None*)

Bases: object

Defines where in images there might by Personally Identifiable information

exception idiscore.image_processing.**PixelDataProcessorException**

Bases: *IDISCoreError*

class idiscore.image_processing.**PixelProcessor**(*location_list:* PIILocationList)

Bases: object

Finds and removes burned-in sensitive information in images

Notes

Responsibilities:

- Checking whether a dataset needs cleaning of its pixel data
- Checking whether redaction can be performed
- Actually performing the blackout

clean_pixel_data(*dataset: Dataset*) → Dataset

Remove pixel data that needs cleaning and mark the dataset as safe

If this dataset does not look suspicious it will not be returned unchanged

Raises
PixelDataProcessorException – If pixel data needs cleaning but no information can be found

get_locations(*dataset: Dataset*) → List[*PIILocation*]

Get all locations with person information in the current dataset

Raises
PixelDataProcessorException – When locations cannot be found properly

static needs_cleaning(*dataset: Dataset*) → bool

Whether this dataset should be rejected as unsafe without cleaning

Made this into a separate method as for many DICOM datasets you can reasonably skip image processing altogether.

Raises
PixelDataProcessorException – When it cannot be determined whether this dataset needs cleaning or not. Usually due to missing DICOM elements

class idiscore.image_processing.**SquareArea**(*origin_x: int*, *origin_y: int*, *width: int*, *height: int*)

Bases: object

A 2D square in pixel coordinates

height: int

origin_x: int

origin_y: int

width: int

3.5.10 idiscore.insertions module

Common DICOM elements you might like to insert into deidentified datasets

This includes the insertions from DICOM PS3.15 E1-1.6:

The attribute Patient Identity Removed (0012,0062) shall be replaced or added to the dataset with a value of YES, and one or more codes from CID 7050 "De-identification Method" corresponding to the profile and options used shall be added to De-identification Method Code Sequence (0012,0064). A text string describing the method used may also be inserted in or added to De-identification Method (0012,0063), but is not required.

idiscore.insertions.**get_deidentification_method**(*method: str = 'idiscore 1.0.3'*) → DataElement

Create the element (0012,0063) - DeIdentificationMethod

A string description of the deidentification method used

Parameters
method (*str, optional*) – String representing the deidentification method used. Defaults to 'idiscore <version>'

idiscore.insertions.**get_idis_code_sequence**(*ruleset_names: List[str]*) → DataElement

Create the element (0012,0064) - DeIdentificationMethodCodeSequence

This sequence specifies what kind of anonymization has been performed. It is quite free form. This implementation uses the following format:

DeIdentificationMethodCodeSequence will contain the code of each official DICOM deidentification profile that was used. Codes are taken from Table CID 7050

Parameters
ruleset_names (*List[str]*) – list of names as defined in nema.E1_1_METHOD_INFO

Returns

Sequence element (0012,0064) - DeIdentificationMethodCodeSequence. Will contain the code of each official DICOM deidentification profile passed

Return type
DataElement

Raises
ValueError – When any name in ruleset_names is not recognized as a standard DICOM rule set

3.5.11 idiscore.nema module

Encodes official NEMA information like Basic Application Level Confidentiality Profile and Options as defined in table E1-1 here: http://dicom.nema.org/medical/dicom/current/output/chtml/part15/sect_E.3.html

This module should model public DICOM information. Any additional information such as default implementations for the action codes should be put in 'rule_sets.py'

class idiscore.nema.**ActionCode**(*key*, *var_name*)

Bases: tuple

key

Alias for field number 0

var_name

Alias for field number 1

class idiscore.nema.**ActionCodes**

Bases: object

NEMA specifications from table E1-1 of what to do with each tag

Modelling these to lessen room for error and to make it easier to write this to disk

```
ALL = {ActionCode(key='C', var_name='CLEAN'), ActionCode(key='D', var_name='DUMMY'),
ActionCode(key='K', var_name='KEEP'), ActionCode(key='U', var_name='UID'),
ActionCode(key='X', var_name='REMOVE'), ActionCode(key='X/D',
var_name='REMOVE_OR_DUMMY'), ActionCode(key='X/Z', var_name='REMOVE_OR_EMPTY'),
ActionCode(key='X/Z/D', var_name='REMOVE_OR_EMPTY_OR_DUMMY'),
ActionCode(key='X/Z/U*', var_name='REMOVE_OR_EMPTY_OR_UID'), ActionCode(key='Z',
var_name='EMPTY'), ActionCode(key='Z/D', var_name='REPLACE_OR_DUMMY')}

CLEAN = ActionCode(key='C', var_name='CLEAN')

DUMMY = ActionCode(key='D', var_name='DUMMY')

EMPTY = ActionCode(key='Z', var_name='EMPTY')

KEEP = ActionCode(key='K', var_name='KEEP')

PER_STRING = {'C': ActionCode(key='C', var_name='CLEAN'), 'D': ActionCode(key='D',
var_name='DUMMY'), 'K': ActionCode(key='K', var_name='KEEP'), 'U':
ActionCode(key='U', var_name='UID'), 'X': ActionCode(key='X', var_name='REMOVE'),
'X/D': ActionCode(key='X/D', var_name='REMOVE_OR_DUMMY'), 'X/Z':
ActionCode(key='X/Z', var_name='REMOVE_OR_EMPTY'), 'X/Z/D': ActionCode(key='X/Z/D',
var_name='REMOVE_OR_EMPTY_OR_DUMMY'), 'X/Z/U*':  ActionCode(key='X/Z/U*',
var_name='REMOVE_OR_EMPTY_OR_UID'), 'Z': ActionCode(key='Z', var_name='EMPTY'),
'Z/D': ActionCode(key='Z/D', var_name='REPLACE_OR_DUMMY')}

REMOVE = ActionCode(key='X', var_name='REMOVE')

REMOVE_OR_DUMMY = ActionCode(key='X/D', var_name='REMOVE_OR_DUMMY')

REMOVE_OR_EMPTY = ActionCode(key='X/Z', var_name='REMOVE_OR_EMPTY')

REMOVE_OR_EMPTY_OR_DUMMY = ActionCode(key='X/Z/D',
var_name='REMOVE_OR_EMPTY_OR_DUMMY')
```

REMOVE_OR_EMPTY_OR_UID = ActionCode(key='X/Z/U*', var_name='REMOVE_OR_EMPTY_OR_UID')

REPLACE_OR_DUMMY = ActionCode(key='Z/D', var_name='REPLACE_OR_DUMMY')

UID = ActionCode(key='U', var_name='UID')

classmethod get_code(*key: str*)

I've got a string. Which action code is this?

class idiscore.nema.**NemaDeidMethodInfo**(*table_header*, *full_name*, *short_name*, *code*)

Bases: `tuple`

code

Alias for field number 3

full_name

Alias for field number 1

short_name

Alias for field number 2

table_header

Alias for field number 0

class idiscore.nema.**RawNemaRuleSet**(*rules: List[Tuple[*TagIdentifier*, *ActionCode*]]*, *name: str*, *code: str*)

Bases: `object`

Defines the action code from table E1-1 for each DICOM identifier

'raw' because an action code is just a string and cannot be applied to a tag. This class defines an intermediate stage in parsing the DICOM confidentiality options. Each identifier has been parsed, but operations have not been assigned

compile(*action_mapping: Dict[*ActionCode*, *Operator*]*) → *RuleSet*

Replace each action code (string) with actual operator (function)

3.5.12 idiscore.operators module

class idiscore.operators.**Clean**(*safe_private: Optional[*SafePrivateDefinition*] = None*, *delta_provider: Optional[*TimeDeltaProvider*] = None*)

Bases: *Operator*

Replace with values of similar meaning known not to contain identifying information and consistent with the VR

'similar meaning' is open to interpretation.

Also handles private tags

apply(*element: DataElement*, *dataset: Optional[Dataset] = None*) → DataElement

Perform this operation on the given element.

Parameters

- **element** (*DataElement*) – The DICOM element to operate on
- **dataset** (*Dataset, optional*) – The DICOM dataset that this element comes from. This can be inspected to determine what to do with element. Should not be changed in any way. Defaults to None

Returns
A new DataElement instance to replace the given element with

Return type
DataElement

Raises

- **ValueError** – When this operation cannot be performed on this element. For example when the data element has a number ValueType but the operation is for a string
- *ElementShouldBeRemoved* – Signals that this element should be removed from the dataset. Operators cannot do this by themselves as they can only operate on the element given

clean_date_time(*element: DataElement*, *dataset: Dataset*) → DataElement

Clean a DICOM date or time

Do this by subtracting a random increment from it

clean_private(*element: DataElement*, *dataset: Dataset*) → DataElement

Clean private DICOM element

is_safe(*element: DataElement*, *dataset: Dataset*) → bool

True if this element is safe according to safe private definition

Raises
SafePrivateError – If for some reason it cannot be determined whether this is safe

name = 'Clean'

static parse_date_time(*value: str*) → Tuple[str, datetime]

Parse DICOM date, datetime or time string

Parameters
value (*str*) – A dicom date datetime or time string

Returns
strptime date format string, parsed datetime instance

Return type
Tuple[str, datetime]

Raises
ValueError – If value cannot be parsed

exception idiscore.operators.**ElementShouldBeRemoved**

Bases: *IDISCoreError*

class idiscore.operators.**Empty**

Bases: *Operator*

Make the content of element empty

apply(*element: DataElement*, *dataset: Optional[Dataset] = None*) → DataElement

Perform this operation on the given element.

Parameters

- **element** (*DataElement*) – The DICOM element to operate on

- **dataset** (*Dataset, optional*) – The DICOM dataset that this element comes from. This can be inspected to determine what to do with element. Should not be changed in any way. Defaults to None

Returns
A new DataElement instance to replace the given element with

Return type
DataElement

Raises

- **ValueError** – When this operation cannot be performed on this element. For example when the data element has a number ValueType but the operation is for a string
- ***ElementShouldBeRemoved*** – Signals that this element should be removed from the dataset. Operators cannot do this by themselves as they can only operate on the element given

name = 'Empty'

class idiscore.operators.**Hash**

Bases: *Operator*

Replace value with an MD5 hash of that value

apply(*element: DataElement*, *dataset: Optional[Dataset] = None*) → DataElement

Perform this operation on the given element.

Parameters

- **element** (*DataElement*) – The DICOM element to operate on
- **dataset** (*Dataset, optional*) – The DICOM dataset that this element comes from. This can be inspected to determine what to do with element. Should not be changed in any way. Defaults to None

Returns
A new DataElement instance to replace the given element with

Return type
DataElement

Raises

- **ValueError** – When this operation cannot be performed on this element. For example when the data element has a number ValueType but the operation is for a string
- ***ElementShouldBeRemoved*** – Signals that this element should be removed from the dataset. Operators cannot do this by themselves as they can only operate on the element given

name = 'Hash'

class idiscore.operators.**HashUID**(*root_uid: Optional[str] = None*)

Bases: *Operator*

Replace element with a valid UID

apply(*element: DataElement*, *dataset: Optional[Dataset] = None*) → DataElement

Perform this operation on the given element.

Parameters

- **element** (*DataElement*) – The DICOM element to operate on
- **dataset** (*Dataset, optional*) – The DICOM dataset that this element comes from. This can be inspected to determine what to do with element. Should not be changed in any way. Defaults to None

Returns
A new DataElement instance to replace the given element with

Return type
DataElement

Raises

- **ValueError** – When this operation cannot be performed on this element. For example when the data element has a number ValueType but the operation is for a string
- ***ElementShouldBeRemoved*** – Signals that this element should be removed from the dataset. Operators cannot do this by themselves as they can only operate on the element given

static ctp_hash_uid(*prefix: str*, *uid: str*)

Implementation of CTP function hashUID(prefix, uid)

Generates a hash of the given UID with the given prefix. Modelled as closely as possible to the java function https://mircwiki.rsna.org/index.php?title=The_CTP_DICOM_Anonymizer#.40hashuid.28root.2CElementName.29

Parameters

- **prefix** (*str*) – DICOM prefix for your organization to prepend in output.
- **uid** (*str*) – original UID

Returns
hashed UID

Return type
str

name = 'HashUID'

class idiscore.operators.**Keep**

Bases: *Operator*

Keep the given element as is. Make no changes

apply(*element: DataElement*, *dataset: Optional[Dataset] = None*) → DataElement

Perform this operation on the given element.

Parameters

- **element** (*DataElement*) – The DICOM element to operate on
- **dataset** (*Dataset, optional*) – The DICOM dataset that this element comes from. This can be inspected to determine what to do with element. Should not be changed in any way. Defaults to None

Returns
A new DataElement instance to replace the given element with

Return type
DataElement

Raises

- **ValueError** – When this operation cannot be performed on this element. For example when the data element has a number ValueType but the operation is for a string
- ***ElementShouldBeRemoved*** – Signals that this element should be removed from the dataset. Operators cannot do this by themselves as they can only operate on the element given

name = 'Keep'

class idiscore.operators.**Operator**

Bases: object

Base class for something that can change a DICOM data element.

Like changing the value, hashing it, removing the entire element, etc. Takes care of input validation, raising exceptions when needed

Notes

Responsibilities

An Operator:

- Can change the single DICOM data element that is fed to it
- Can inspect the dataset that is passed to it
- Can take init arguments and connect to external resources if needed
- Should NOT alter the dataset that is passed to it

apply(*element: DataElement*, *dataset: Optional[Dataset] = None*) → DataElement

Perform this operation on the given element.

Parameters

- **element** (*DataElement*) – The DICOM element to operate on
- **dataset** (*Dataset, optional*) – The DICOM dataset that this element comes from. This can be inspected to determine what to do with element. Should not be changed in any way. Defaults to None

Returns
A new DataElement instance to replace the given element with

Return type
DataElement

Raises

- **ValueError** – When this operation cannot be performed on this element. For example when the data element has a number ValueType but the operation is for a string
- ***ElementShouldBeRemoved*** – Signals that this element should be removed from the dataset. Operators cannot do this by themselves as they can only operate on the element given

name = 'Base Operation'

class idiscore.operators.**Remove**

Bases: *Operator*

Remove the given element completely

apply(*element: DataElement*, *dataset: Optional[Dataset] = None*)

Perform this operation on the given element.

Parameters

- **element** (*DataElement*) – The DICOM element to operate on
- **dataset** (*Dataset, optional*) – The DICOM dataset that this element comes from. This can be inspected to determine what to do with element. Should not be changed in any way. Defaults to None

Returns
A new DataElement instance to replace the given element with

Return type
DataElement

Raises

- **ValueError** – When this operation cannot be performed on this element. For example when the data element has a number ValueType but the operation is for a string
- ***ElementShouldBeRemoved*** – Signals that this element should be removed from the dataset. Operators cannot do this by themselves as they can only operate on the element given

name = 'Remove'

class idiscore.operators.**Replace**

Bases: *Operator*

Replace element with a dummy value

apply(*element: DataElement*, *dataset: Optional[Dataset] = None*) → DataElement

Perform this operation on the given element.

Parameters

- **element** (*DataElement*) – The DICOM element to operate on
- **dataset** (*Dataset, optional*) – The DICOM dataset that this element comes from. This can be inspected to determine what to do with element. Should not be changed in any way. Defaults to None

Returns
A new DataElement instance to replace the given element with

Return type
DataElement

Raises

- **ValueError** – When this operation cannot be performed on this element. For example when the data element has a number ValueType but the operation is for a string
- ***ElementShouldBeRemoved*** – Signals that this element should be removed from the dataset. Operators cannot do this by themselves as they can only operate on the element given

name = 'Replace'

class idiscore.operators.**SetFixedValue**(*value: Union[str, int, object]*)

Bases: *Operator*

Replace element with a fixed value from a list of tag-value pairs

apply(*element: DataElement*, *dataset: Optional[Dataset] = None*) → DataElement

Perform this operation on the given element.

Parameters

- **element** (*DataElement*) – The DICOM element to operate on
- **dataset** (*Dataset, optional*) – The DICOM dataset that this element comes from. This can be inspected to determine what to do with element. Should not be changed in any way. Defaults to None

Returns
A new DataElement instance to replace the given element with

Return type
DataElement

Raises

- **ValueError** – When this operation cannot be performed on this element. For example when the data element has a number ValueType but the operation is for a string
- *ElementShouldBeRemoved* – Signals that this element should be removed from the dataset. Operators cannot do this by themselves as they can only operate on the element given

name = 'SetFixedValue'

class idiscore.operators.**TimeDeltaProvider**

Bases: object

Generates a random shift in time to use when cleaning dates.

Returns the same output for data sets in the same study

static extract_key(*dataset: Dataset*) → str

Extracts a key from dataset. Data sets with the same key will be given the same delta

Raises
ValueError – If key cannot be generated

static generate_random_delta() → timedelta

Anything from 0 up to 5 years and 23:59 and 59 seconds

get_delta(*dataset: Dataset*) → timedelta

Returns the same delta if a dataset belongs to a series already seen

If series cannot be determined, return random delta

3.5.13 idiscore.privateprocessing module

Classes and methods for handling private DICOM elements

Is a private tag is safe to keep? This can not be answered with regular rules of the form tag -> operation. Sometimes you need to inspect the entire dataset, for example to check modality or vendor.

class idiscore.private_processing.**SafePrivateBlock**(*tags: Iterable[Union[*PrivateBlockTagIdentifier*, str]]*, *criterion: Optional[Callable[[Dataset], bool]] = None*, *comment: str = ''*)

Bases: `object`

Defines when one or more private DICOM elements can be considered 'safe'

Safe as in 'not containing personally identifiable information'

get_safe_private_tags(*dataset: Dataset*) → Set[*TagIdentifier*]

The private tags that are safe to keep, given this dataset

Raises
CriterionException – If no True or False response can be given for this dataset

tags_are_safe(*dataset: Dataset*) → bool

True if these private tags are safe to keep in this dataset

static to_tag_identifier(*tag_or_string: Union[*PrivateBlockTagIdentifier*, str]*) → *PrivateBlockTagIdentifier*

Cast any string to tag identifier. If already a TagIdentifier do nothing

Return type
TagIdentifier

Raises
ValueError – if tag is string and is not in the correct format

class idiscore.private_processing.**SafePrivateDefinition**(*blocks: List[*SafePrivateBlock*]*)

Bases: `object`

Holds all information on which private tags can be considered safe

Contains one or more SafePrivateBlocks

is_safe(*element: DataElement*, *dataset: Dataset*) → bool

True if the given private element in the given dataset is safe to keep

Raises
SafePrivateError – If for some reason it cannot be determined whether this is safe

safe_identifiers(*dataset: Dataset*) → List[*TagIdentifier*]

All tags that are safe to keep given this dataset

Raises
SafePrivateError – If safe identifiers cannot be determined

3.5.14 idiscore.rule_sets module

Common sets of rules to deidentify multiple dicom elements

Contains default implementations of the DICOM standard deidentification profiles and options and other useful sets

class idiscore.rule_sets.**DICOMRuleSets**(*action_mapping: Optional[Dict[*ActionCode*,* Operator*]] = None*)

Bases: object

Holds the rule sets for DICOM deidentification basic profile and options

These are lists of rules that implement the actions designated in table E3

Notes

More information on profile and options found here: http://dicom.nema.org/medical/dicom/current/output/chtml/part15/sect_E.3.html

3.5.15 idiscore.rules module

class idiscore.rules.**Rule**(*identifier: Union[*TagIdentifier*, BaseTag]*, *operation:* Operator)

Bases: object

Defines what to do with a single DICOM element or single group of elements

as_human_readable() → str

matches(*element: DataElement*) → bool

True if this rule matches the given DICOM element

number_of_matchable_tags() → int

The number of distinct DICOM tags that this rule could match

class idiscore.rules.**RuleSet**(*rules: Iterable[*Rule*]*, *name: str = 'RuleSet'*)

Bases: object

Defines what to do to one or more DICOM tags

Models part of a deidentification procedure, such as the Basic Application Level Confidentiality Options in DICOM (e.g. Retain Safe Private Option)

as_dict() → Dict[*TagIdentifier*, *Rule*]

as_human_readable_list() → str

All rules in this set sorted by tag name

get_rule(*element: DataElement*) → Optional[*Rule*]

The most specific rule for the given DICOM element, or None if not found

Returns

- *Rule* – Most specific rule for the given DICOM tag
- *None* – If no rule matches the given DICOM tag

Notes

It is possible for multiple rules to match. Lookup is always done from specific to general. For example, when getting a rule for element with tag (0010,0010):

- A rule for (0010,0010) is preferred over (0010,00xx)
- A rule for (0010,00xx) is preferred over (0010,xx10)
- A rule for (0010,xx10) is preferred over (xxxx,0010)

Generality is determined by the *number_of_matchable_tags()* function of each rule. The more tags that could be matched, the more general the rule is

static is_single_tag_rule(*rule:* Rule) → bool

Targets only a single DICOM tag

remove(*rule:* Rule)

Remove the given rule from this set

Raises
KeyError – If rule is not in this set

property rules: Set[*Rule*]

All rules in this list

static tag_to_key(*tag: BaseTag*) → str

Represent tag as single 8 char hex string like '00100010'

This is the format used as dict key internally

3.5.16 idiscore.settings module

3.5.17 idiscore.templates module

Jinja templates. Putting these in a separate module because indentation is difficult when inlining templates inside classes and functions

idiscore.templates.**make_h1**(*text*)

idiscore.templates.**make_h2**(*text*)

idiscore.templates.**make_h3**(*text*)

3.5.18 idiscore.validation module

3.6 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

3.6.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/sjoerdk/idiscore/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

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

Implement Features

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

Write Documentation

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

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/sjoerdk/idiscore/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.6.2 Get Started!

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

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

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

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

```
$ mkvirtualenv idiscore
$ cd idiscore/
$ python setup.py develop
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 idiscore tests
$ python setup.py test or pytest
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

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

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

7. Submit a pull request through the GitHub website.

3.6.3 Pull Request Guidelines

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

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.8, and for PyPy. Check https://github.com/sjoerdk/idiscore/actions?query=workflow%3Abuild and make sure that the tests pass for all supported Python versions.

3.6.4 Tips

To run a subset of tests:

```
$ pytest tests.test_idiscore
```

3.6.5 Development

Some notes

- idiscore is python-only. We recommend pycharm as an editor
- Work via pull requests: clone the idiscore repo, make changes and make a pull request

3.6.6 Code quality

All code must conform to flake8. And black Build will fail for non-conformant code. Either run flake8 and black yourself (in repo root folder, type *flake8 idiscore tests*, and 'black .') or install the pre-commit hooks:

```
$ python3 -m pip install pre-commit
$ python3 -m pre-commit install
```

This will run black and flake8 automatically before any commit

3.7 History

3.7.1 1.1.0 (2022-09-15)

- Stoped internal deepcopy DICOM files, improving performance and reducing IO issues
- Adopted PEP517 for package management. Using poetry now
- Packaging: push to pypi is now only done on github publish.

3.7.2 1.0.0 (2020-08-20)

- Deidentification implementing standard DICOM confidentiality profile and options
- Basic imagedata processing
- Support for safe private tags
- Documentation
- Line coverage over 90%

3.7.3 0.3.1 (2020-08-02)

- Alpha development

3.7.4 0.1.0 (2020-06-02)

- First release on PyPI.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

i

INDEX

N

O

P

R

S

T

U

V

W