deid Documentation

Release 0.3.1

Vanessa Sochat

Nov 22, 2022

API REFERENCE

This is the developer documentation (meaning docstrings) for deid. For user guides and tutorials see the main documentation. To see the code, head over to the repository.

CHAPTER

ONE

SUPPORT

- For **bugs and feature requests**, please use the issue tracker.
- For **contributions**, visit Caliper on Github.

CHAPTER

TWO

RESOURCES

GitHub Repository The code on GitHub.

Documentation The main user guide.

Pydicom The core pydicom to read dicom in Python.

2.1 deid.config package

2.1.1 Submodules

2.1.2 deid.config.standards module

2.1.3 deid.config.utils module

deid.config.utils.**add_section**(*config*, *section*, *section_name=None*)
add section will add a section (and optionally) section name to a config

Parameters

- **config**(*the config (dict) parsed thus far*) –
- **section**(*the section name to add*) –
- **section_name**(*an optional name, added as a level*) –

deid.config.utils.**find_deid**(*path=None*)
find_deid is a helper function to load_deid to find a deid file.

It can be in a folder, or return the path provided if it is the file.

Parameters **path** (*a path on the filesystem. If not provided, will assume PWD.*) –

deid.config.utils.**get_deid**(*tag=None*, *exit_on_fail=True*, *quiet=False*, *load=False*)
get deid is intended to retrieve the full path of a deid file provided with the software, based on a tag. For example, under deid/data if a file is called "deid.dicom", the tag would be "dicom".

Parameters

- **tag** (*the text that comes after deid to indicate the tag of the file in deid/data*) –
- **exit_on_fail** (*if None is an acceptable return value, this should be set to False*) – (default is True).

- **quiet** (*Default False. If None is acceptable, quiet can be set to True*) –
- **load** (*also load the deid, if resulting path (from path or tag) is not None*) –

deid.config.utils.**load_combined_deid**(*deids*)
load one or more deids, either based on a path or a tag

Parameters **deids** (*should be a custom list of deids*) –

deid.config.utils.**load_deid**(*path=None*)
Load_deid will return a loaded in (user) deid configuration file.

This can be used to update a default config.json. If a file path is specified, it is loaded directly. If a folder is specified, we look for a deid file in the folder. If nothing is specified, we assume the user wants to load a deid file in the present working directory. If the user wants to have multiple deid files in a directory, this can be done with an extension that specifies the module, eg;

deid.dicom deid.nifti

Parameters **path** (*a path to a deid file*) –

Returns **config**

Return type a parsed deid (dictionary) with valid sections

deid.config.utils.**parse_config_action**(*section*, *line*, *config*, *section_name=None*)
add action will take a line from a deid config file, a config (dictionary), and an active section name (eg header) and add an entry to the config file to perform the action.

Parameters

- **section** (*a valid section name from the deid config file*) –
- **line** (*the line content to parse for the section/action*) –
- **config** (*the growing/current config dictionary*) –
- **section_name** (*optionally, a section name*) –

deid.config.utils.**parse_filter_group**(*spec*)
given the specification (a list of lines) continue parsing lines until the filter group ends, as indicated by the start of a new LABEL, (case 1), the start of a new section (case 2) or the end of the spec file (case 3). Returns a list of members (lines) that belong to the filter group. The list (by way of using pop) is updated in the calling function.

Parameters **spec** (*unparsed lines of the deid recipe file*) –

deid.config.utils.**parse_format**(*line*)
given a line that starts with FORMAT, parse the file.

This means checking the format of the file and checking that it is supported. If not, exit on error. If yes, return the format.

Parameters **line** (*the line that starts with format.*) –

deid.config.utils.**parse_group_action**(*section*, *line*, *config*, *section_name*)
parse a group action, either FIELD or SPLIT, which must belong to either a fields or values section.

Parameters

- **section** (*a valid section name from the deid config file*) –
- **line** (*the line content to parse for the section/action*) –

- **config** (*the growing/current config dictionary*) –
- **section_name** (*optionally, a section name*) –

deid.config.utils.**parse_label**(*section*, *config*, *section_name*, *members*, *label=None*)
Add a named label to the filter section, including one or more criteria

Parameters

- **section** (*the section name (e.g., header) must be one in sections*) –
- **config** (*the config (dictionary) parsed thus far*) –
- **section_name** (*an optional name for a section*) –
- **members** (*the lines belonging to the section/section_name*) –
- **label** (*an optional name for the group of commands*) –

deid.config.utils.**parse_member**(*members*, *operator=None*)
a parsing function for a filter member. Will return a single member with fields, values, and an operator. In the case of multiple and/or statements that are chained, will instead return a list.

2.1.4 Module contents

class deid.config.**DeidRecipe**(*deid=None*, *base=False*, *default_base='dicom'*)
Bases: object

Create a deid recipe to filter and perform operations on a dicom header.

Usage typically looks like:

deid = 'dicom.deid' recipe = DeidRecipe(deid)

If deid is None, the default provided by the application is used.

Parameters

- **deid** (*the deid recipe (or recipes) files to use. If more than one*) – is provided, should be done in order of preference for load (later in the list overrides earlier loaded).
- **base** (*if True, load a default base (default_base) before custom*) –
- **default_base** (*the default base to load if "base" is True*) –

get_actions(*action=None*, *field=None*)
Get deid actions to perform on a header, or a subset based on a type

A header action is a list with the following: {'action': 'REMOVE', 'field': 'AssignedLocation'},

Parameters

- **action** (*if not None, filter to action specified*) –
- **field** (*if not None, filter to field specified*) –

get_fields_lists(*name=None*)
Return a values list by name

get_filters(*name=None*)
Return all filters for a deid recipe, or a set based on a name

get_format()
Return the format of the loaded deid, if one exists

get_values_lists(*name=None*)
Return a values list by name

has_actions()

has_fields_lists()

has_values_lists()

listof(*section*)
Return a list of keys for a section

load(*deid*)
Load a deid recipe into the object.

If a deid configuration is already defined, append to that.

ls_fieldlists()

ls_filters()

ls_valuelists()

2.2 deid.data package

2.2.1 Module contents

deid.data.**get_dataset**(*dataset=None*)
Get a dataset by name.

get_dataset will return some data provided by the application, based on a user-provided label. In the future, we can add https endpoints to retrieve online datasets.

2.3 deid.dicom.actions package

2.3.1 Submodules

2.3.2 deid.dicom.actions.jitter module

deid.dicom.actions.jitter.**jitter_timestamp**(*field*, *value*)
Jitter a timestamp "field" by number of days specified by "value"

The value can be positive or negative. This function is grandfathered into deid custom funcs, as it existed before they did. Since a custom func requires an item, we have a wrapper above to support this use case.

Parameters

- **field**(*the field with the timestamp*) –
- **value**(*number of days to jitter by. Jitter bug!*) –

deid.dicom.actions.jitter.**jitter_timestamp_func**(*item*, *value*, *field*, ***kwargs*)
A wrapper to jitter_timestamp so it works as a custom function.

2.3.3 deid.dicom.actions.uids module

deid.dicom.actions.uids.**basic_uuid**(*item*, *value*, *field*, ***kwargs*)
A basic function to replace a field with a uuid.uuid4() string

deid.dicom.actions.uids.**dicom_uuid**(*item*, *value*, *field*, *dicom*, ***kwargs*)
Generate a dicom uid that better conforms to the dicom standard.

deid.dicom.actions.uids.**pydicom_uuid**(*item*, *value*, *field*, ***kwargs*)
Use pydicom to generate the UID. Optional kwargs include:

prefix (str): provide a custom prefix stable_remapping (bool): if true, use the orignal value for entropy. This ensures stability across different runs that use the same UID.

The prefix must match '^(0|[1-9][0-9]*)(.(0|[1-9][0-9]*))*.$'

deid.dicom.actions.uids.**suffix_uuid**(*item*, *value*, *field*, ***kwargs*)
Return the same field, with a uuid suffix.

Provided in docs: https://pydicom.github.io/deid/examples/func-replace/

2.3.4 Module contents

2.4 deid.dicom.pixels package

2.4.1 Submodules

2.4.2 deid.dicom.pixels.clean module

class deid.dicom.pixels.clean.**DicomCleaner**(*output_folder=None*, *add_padding=False*, *margin=3*, *deid=None*, *font=None*, *force=True*)
Bases: object

Clean a dicom file of burned pixels.

take an input dicom file, check for burned pixels, and then clean, with option to save / output in multiple formats. This object should map to one dicom file, and the usage flow is the following: cleaner = DicomCleaner() summary = cleaner.detect(dicom_file)

cleaner.clean()

clean(*fix_interpretation: bool = True*, *pixel_data_attribute: str = 'PixelData'*) → Optional[numpy.ndarray[Any, numpy.dtype[ScalarType]]]

default_font()
Get the default font to use for a title.

define the font style for saving png figures if a title is provided

detect(*dicom_file*)
Initiate the cleaner for a new dicom file.

get_figure(*show=False*, *image_type='cleaned'*, *title=None*)
Get a figure for an original or cleaned image.

If the image was already clean, it is simply a copy of the original. If show is True, plot the image. If a 4d image is discovered, we use randomly choose a slice.

save_animation(*output_folder=None*, *image_type='cleaned'*, *title=None*)
Save an original or cleaned animation of a dicom.

If there are not enough frames, then save_png should be used instead.

save_dicom(*output_folder=None*, *image_type='cleaned'*)
Save a cleaned dicom to disk.

We expose an option to save an original (change image_type to "original" to be consistent, although this is not incredibly useful given it would duplicate the original data.

save_png(*output_folder=None*, *image_type='cleaned'*, *title=None*)
Save an original or cleaned dicom as png to disk.

Default image_format is "cleaned" and can be set to "original." If the image was already clean (not flagged) the cleaned image is just a copy of original. If a 4d image is provided, we save the dimension specified (or if not provided, a randomly chosen dimension).

deid.dicom.pixels.clean.**clean_pixel_data**(*dicom_file*, *results: dict*, *fix_interpretation: bool = True*, *pixel_data_attribute: str = 'PixelData'*)
Clean a dicom file.

take a dicom image and a list of pixel coordinates, and return a cleaned file (if output file is specified) or simply plot the cleaned result (if no file is specified)

Parameters

- **dicom_file** (*(str or FileDataset instance) Dicom file to clean*) –
- **results** (*Result of the .has_burned_pixels() method*) –
- **fix_interpretation** (*fix the photometric interpretation if found off*) –
- **pixel_data_attribute** (*PixelData attribute name in the dicom file*) –

2.4.3 deid.dicom.pixels.detect module

deid.dicom.pixels.detect.**evaluate_group**(*flags*)
Evaluate group will take a list of flags (e.g.,

[True, and, False, or, True]

And read through the logic to determine if the image result is to be flagged. This is how we combine a set of criteria in a group to come to a final decision.

deid.dicom.pixels.detect.**extract_coordinates**(*dicom*, *field*)
Given a field that is provided for a dicom, extract coordinates

deid.dicom.pixels.detect.**has_burned_pixels**(*dicom_files*, *force: bool = True*, *deid: Optional[*deid.config.DeidRecipe*] = None*)
Determine if a dicom file has burned pixels.

has_burned_pixels is an entrypoint for has_burned_pixels_multi (for multiple images) or has_burned_pixels_single (for one detailed repor) We will use the MIRCTP criteria (see ref folder with the original scripts used by CTP) to determine if an image is likely to have PHI, based on fields in the header alone. This script does NOT perform pixel cleaning, but returns a dictionary of results (for multi) or one detailed result (for single)

2.4.4 Module contents

2.5 deid.dicom package

2.5.1 Subpackages

2.5.2 Submodules

2.5.3 deid.dicom.fields module

class deid.dicom.fields.**DicomField**(*element*, *name*, *uid*, *is_filemeta=False*)
Bases: object

A dicom field.

A dicom field holds the element, and a string that represents the entire nested structure (e.g., SequenceName__CodeValue).

name_contains(*expression*)
Determine if a name contains a pattern or expression.

Use re to search a field for a regular expression, meaning the name, the keyword (nested) or the string tag.

name.lower: includes nested keywords (e.g., Sequence_Child) self.tag: is the string version of the tag self.element.name: is the human friendly name "Sequence Child" self.element.keyword: is the name without nesting "Child"

select_matches(*expression*)
Determine whether the element has a specific selected attribute

property stripped_tag
Return the stripped element tag

property tag
Return a string of the element tag.

value_contains(*expression*)
Use re to search a field value for a regular expression

deid.dicom.fields.**expand_field_expression**(*field*, *dicom*, *contenders=None*)
Get a list of fields based on an expression.

If no expression found, return single field. Options for fields include:

endswith: filter to fields that end with the expression startswith: filter to fields that start with the expression contains: filter to fields that contain the expression select: filter based on DICOM element properties allfields: include all fields exceptfields: filter to all fields except those listed (| separated)

Returns: a list of DicomField objects

deid.dicom.fields.**extract_item**(*item*, *prefix=None*, *entry=None*)
Extract values from a dicom sequence depending on the type.

A helper function to extract sequence, will extract values from a dicom sequence depending on the type.

Parameters **item**(*an item from a sequence.*) –

deid.dicom.fields.**extract_sequence**(*sequence*, *prefix=None*)
Extract a sequence recursively.

return a pydicom.sequence.Sequence recursively as a flattened list of items. For example, a nested FieldA and FieldB would return as:

{'FieldA__FieldB': '111111'}

Parameters

- **sequence** (*the sequence to extract, should be pydicom.sequence.Sequence*) –
- **prefix** (*the parent name*) –

deid.dicom.fields.**get_fields**(*dicom*, *skip=None*, *expand_sequences=True*, *seen=None*)
Expand all dicom fields into a list.

Each entry is a DicomField. If we find a sequence, we unwrap it and represent the location with the name (e.g., Sequence__Child)

2.5.4 deid.dicom.filter module

deid.dicom.filter.**apply_filter**(*dicom*, *field*, *filter_name*, *value*)
essentially a switch statement to apply a filter to a dicom file.

Parameters

- **dicom** (*the pydicom.dataset Dataset (pydicom.read_file)*) –
- **field** (*the name of the field to apply the filter to*) –
- **filer_name** (*the name of the filter to apply (e.g., contains)*) –
- **value** (*the value to set, if filter_name is valid*) –

deid.dicom.filter.**compareBase**(*self*, *field*, *expression*, *func*, *ignore_case=True*)
Search a field for an expression.

compareBase takes either re.search (for contains) or re.match (for matches) and returns True if the given regular expression is contained or matched

deid.dicom.filter.**contains**(*self*, *field*, *expression*)
Determine if a field value contains an expression.

contains returns true if the value of the identifier contains the the string argument anywhere within it; otherwise, it returns false.

deid.dicom.filter.**empty**(*self*, *field*)
Determine if the value is empty.

Empty returns True if the value is found to be "". If the field is not present for the dicom, then we return False (missing != empty)

deid.dicom.filter.**endsWith**(*self*, *field*, *term*)
Determine if a field value ends with an expression.

endsWith returns true if the value of the identifier ends with the string argument; otherwise, it returns false.

deid.dicom.filter.**equals**(*self*, *field*, *term*)
returns true if the value of the identifier exactly equals the string argument; otherwise, it returns false.

deid.dicom.filter.**equalsBase**(*self*, *field*, *term*, *ignore_case=True*, *not_equals=False*)
base of equals, with variable for ignore case (default True)

deid.dicom.filter.**matches**(*self*, *field*, *expression*)
Determine if a field value matches an expression.

matches returns true if the value of the identifier matches the regular expression specified in the string argument; otherwise, it returns false.

deid.dicom.filter.**missing**(*self*, *field*)
Determine if the dicom is missing a field.

Missing returns True if the dicom is missing the field entirely This means that the entire field is None

deid.dicom.filter.**notContains**(*self*, *field*, *expression*)
Determine if a field value does not contain an expression.

notContains returns true if the value of the identifier does not contain the the string argument anywhere within it;

deid.dicom.filter.**notEquals**(*self*, *field*, *term*)

deid.dicom.filter.**startsWith**(*self*, *field*, *term*)
Determine if a field value starts with an expression.

startsWith returns true if the value of the identifier starts with the string argument; otherwise, it returns false.

2.5.5 deid.dicom.groups module

deid.dicom.groups.**extract_fields_list**(*dicom*, *actions*, *fields=None*)
Given a list of actions for a named group (a list) extract values from the dicom based on the list of actions provided. This function always returns a list intended to update some lookup to be used to further process the dicom.

deid.dicom.groups.**extract_values_list**(*dicom*, *actions*, *fields=None*)
Given a list of actions for a named group (a list) extract values from the dicom based on the list of actions provided. This function always returns a list intended to update some lookup to be used to further process the dicom.

2.5.6 deid.dicom.header module

deid.dicom.header.**get_identifiers**(*dicom_files*, *force=True*, *config=None*, *strip_sequences=False*, *remove_private=False*, *disable_skip=False*, *expand_sequences=True*)
Extract all identifiers from a dicom image.

This function returns a lookup by file name, where each value indexed includes a dictionary of nested fields (indexed by nested tag).

Parameters

- **dicom_files**(*the dicom file(s) to extract from*) –
- **force**(*force reading the file (default True)*) –
- **config**(*if None, uses default in provided module folder*) –
- **strip_sequences**(*if True, remove all sequences*) –
- **remove_private**(*remove private tags*) –
- **disable_skip**(*do not skip over protected fields*) –

- **expand_sequences** (*if True, expand sequences. otherwise, skips*) –

deid.dicom.header.**remove_private_identifiers**(*dicom_files*, *save=True*, *overwrite=False*, *output_folder=None*, *force=True*)

Remove private identifiers.

remove_private_identifiers is a wrapper for the simple call to dicom.remove_private_tags, it simply reads in the files for the user and saves accordingly

deid.dicom.header.**replace_identifiers**(*dicom_files*, *ids=None*, *deid=None*, *save=False*, *overwrite=False*, *output_folder=None*, *force=True*, *config=None*, *strip_sequences=False*, *remove_private=False*, *disable_skip=False*)

Replace identifiers.

replace identifiers using pydicom, can be slow when writing and saving new files. If you want to replace sequences, they need to be extracted with get_identifiers and expand_sequences to True.

2.5.7 deid.dicom.parser module

class deid.dicom.parser.**DicomParser**(*dicom_file*, *recipe=None*, *config=None*, *force=True*, *disable_skip=False*)

Bases: object

Parse a dicom, performing one or more actions on fields.

A dicom parser serves as a cache to read in all fields from a dicom file. For each, we store the element and child elements

add_field(*field*, *value*)
Add a field to the dicom.

If it's already present, update the value.

blank_field(*field*)
Blank a field

define(*name*, *value*)
Add a function or variable to the lookup for later usage.

This can be used for functions, lists, or variables.

delete_field(*field*)
Delete a field from the dicom.

We do this by way of parsing all nested levels of a tag into actual tags, and deleting the child node.

property excluded_from_deletion
Return once-evaluated list of fields that are not removed by REMOVE ALL or REMOVE SomeField, as they later have to be changed by REPLACE / JITTER That allows whitelisting fields from REMOVE ALL/SomeField to change them if needed (i.e. obfuscation)

find_by_name(*name*)
Find fields by name.

Given a string, find all field objects that contain the name. Name can correspond to:

- a string of the tag, with or without the parens and comma/space
- a keyword
- a field name

find_by_values(*values*)
Find fields by values.

Given a list of values, find fields in the dicom that contain any of those values, as determined by a regular expression search.

get_fields(*expand_sequences=True*)
expand all dicom fields into a list, where each entry is a DicomField. If we find a sequence, we unwrap it and represent the location with the name (e.g., Sequence__Child)

get_nested_field(*field*, *return_parent=False*)
Retrieve a nested field.

Based on a DicomField, return the one referenced in self.dicom. If a delete is needed, then the parent should be returned as well.

property keep
Return a list of fields to keep original, as defined by all KEEP actions in recipe Those fields are not impacted by REPLACE/JITTER actions

load(*dicom_file*, *force=True*)
Load the dicom file.

Ensure that the dicom file exists, and use full path. Here we load the file, and save the dicom, dicom_file, and dicom_name.

parse(*strip_sequences=False*, *remove_private=False*)
Parse the dicom.

The parse action corresponds to iterating through fields, and for each one, saving a data structure with the full element, the string (with nested representation of the keywords) and the tag. We want to save all three in a flat list that is easy to search over, and also build up actions for the lookup on the first parsing.

perform_action(*field*, *value*, *action*, *filemeta=False*)
Perform an action on a field.

perform action takes an action (dictionary with field, action, value) and performs the action on the loaded dicom.

Parameters

- **field**(*a field for expand*) –
- **value**(*field value*) –
- **action**(*the action from the parsed deid to take*) – "field" (eg, PatientID) the header field to process "action" (eg, REPLACE) what to do with the field "value": if needed, the field from the response to replace with

remove_private()
Remove private tags from the loaded dicom

replace_field(*field*, *value*)
Replace a value in a field.

This uses the same function as ADD, but likely the dicom has the value.

reset_preamble()
reset the preamble

save(*filename*, *overwrite=False*)
Save a dicom to file.

property skip
Return a list of fields to skip, as defined in the self.config

2.5.8 deid.dicom.tags module

deid.dicom.tags.**add_tag**(*identifier*, *VR='ST'*, *VM=None*, *name=None*, *keyword=None*)
Add tag will take a string for a tag (e.g.,) and define a new tag for it. By default, we give the type "Short Text."

deid.dicom.tags.**find_tag**(*term*, *VR=None*, *VM=None*, *retired=False*)
find_tag will search over tags in the DicomDictionary and return the tags found to match some term.

deid.dicom.tags.**get_private**(*dicom*)
get private tags

Parameters **dicom**(*the pydicom.dataset Dataset (pydicom.read_file)*)–

deid.dicom.tags.**get_tag**(*field*)
get_tag will return a dictionary with tag indexed by field. For each entry, a dictionary lookup is included with VR.

Parameters **field** (*the keyword to get tag for, eg "PatientIdentityRemoved"*)–

deid.dicom.tags.**has_private**(*dicom*)
has_private will return True if the header has private tags

Parameters **dicom**(*the pydicom.dataset Dataset (pydicom.read_file)*)–

deid.dicom.tags.**remove_sequences**(*dicom*)
remove sequences from a dicom by removing the associated tag. We use dicom.iterall() to get all nested sequences.

Parameters **dicom**(*the loaded dicom to remove sequences*)–

deid.dicom.tags.**update_tag**(*dicom*, *field*, *value*)
update tag will update a value in the header, if it exists if not, nothing is added. This check is the only difference between this function and change_tag. If the user wants to add a value (that might not exist) the function add_tag should be used with a private identifier as a string.

Parameters

- **dicom**(*the pydicom.dataset Dataset (pydicom.read_file)*)–
- **field**(*the name of the field to update*)–
- **value**(*the value to set, if name is a valid tag*)–

2.5.9 deid.dicom.utils module

deid.dicom.utils.**get_files**(*contenders*, *check=True*, *pattern=None*, *force=False*, *tempdir=None*)
Get a generator for files.

get_files will take a list of single dicom files or directories, and return a generator that yields complete paths to all files

Parameters

- **contenders**(*a list of files or directories (contenders!)*)–
- **check** (*boolean to indicate if we should validate dicoms (default True)*)–

- **pattern** (*A pattern to use with fnmatch. If None, * is used*) –
- **force** (*force reading of the files, if some headers invalid.*) – Not recommended, as many non-dicom will come through

deid.dicom.utils.**load_dicom**(*dcm_file*)

deid.dicom.utils.**save_dicom**(*dicom*, *dicom_file*, *output_folder=None*, *overwrite=False*)
Save a dicom file to an output folder.

We make sure to not overwrite unless the user has enforced it

Parameters

- **dicom** (*the pydicon Dataset to save*) –
- **dicom_file** (*the path to the dicom file to save (we only use basename)*) –
- **output_folder** (*the folder to save the file to*) –
- **overwrite** (*overwrite any existing file? (default is False)*) –

2.5.10 deid.dicom.validate module

deid.dicom.validate.**validate_dicoms**(*dcm_files*, *force=False*)
Validate that dicom files can open and return valid set.

validate dicoms will test opening one or more dicom files, and return a list of valid files.

Parameters **dcm_files** (*one or more dicom files to test*) –

2.5.11 Module contents

2.6 deid.logger package

2.6.1 Submodules

2.6.2 deid.logger.message module

class deid.logger.message.**DeidMessage**(*MESSAGELEVEL=None*)
Bases: object

abort(*message*)

addColor(*level*, *text*)
Add color to the prompt.

addColor to the prompt (usually prefix) if terminal supports, and specified to do so

custom(*prefix*, *message*, *color='\x1b[95m'*)

debug(*message*)

emit(*level*, *message*, *prefix=None*, *color=None*)
Emit a message.

Emit is the main function to print the message optionally with a prefix

Parameters

- **level**(*the level of the message*)–
- **message**(*the message to print*)–
- **prefix**(*a prefix for the message*)–

emitError(*level*)
Determine if we should emit an error message to stderr.

This includes all levels but INFO and QUIET

emitOutput(*level*)
Determine if a level should print to stdout.

error(*message*)

exit(*message*, *return_code=1*)

flag(*message*)

get_logs(*join_newline=True*)
Return the complete logging history.

info(*message*)

isEnabledFor(*messageLevel*)
Check if a messageLevel is enabled to emit a level

is_quiet()
is_quiet returns true if the level is under 1

log(*message*)

newline()

show_progress(*iteration*, *total*, *length=40*, *min_level=0*, *prefix=None*, *carriage_return=True*, *suffix=None*, *symbol=None*)
Create a terminal progress bar. :param iteration: current iteration (Int) :param total: total iterations (Int) :param length: character length of bar (Int)

table(*rows*, *col_width=2*)
Print a table of entries.

Table will print a table of entries. If the rows is a dictionary, the keys are interpreted as column names. if not, a numbered list is used.

useColor()
Determine if we should add color to a print.

useColor will determine if color should be added to a print. Will check if being run in a terminal, and if has support for asci

verbose(*message*)

verbose1(*message*)

verbose2(*message*)

verbose3(*message*)

warning(*message*)

write(*stream*, *message*)
Write a message to a stream.

deid.logger.message.**convert2boolean**(*arg*)
Convert envars to boolean.

convert2boolean is used for environmental variables that must be returned as boolean

deid.logger.message.**get_logging_level**()
Get the logging level.

get_logging_level will configure a logging to standard out based on the user's selected level, which should be in an environment variable called MESSAGELEVEL. if MESSAGELEVEL is not set, the maximum level (5) is assumed (all messages).

deid.logger.message.**get_user_color_preference**()

2.6.3 deid.logger.progress module

clint.textui.progress

A derivation of clint version, to not introduce a dependency and add custom functionality. Credit to base code goes to https://github.com/kennethreitz/clint/blob/master/clint/textui/progress.py

class deid.logger.progress.**ProgressBar**(*label=''*, *width=32*, *hide=None*, *empty_char=' '*, *filled_char='-'*, *expected_size=None*, *every=1*)
Bases: object

done()

format_time(*seconds*)

show(*progress*, *count=None*)

deid.logger.progress.**bar**(*it*, *label=''*, *width=32*, *hide=None*, *empty_char=' '*, *filled_char='-'*, *expected_size=None*, *every=1*)
Progress iterator. Wrap your iterables with it.

2.6.4 Module contents

2.7 deid.main package

2.7.1 Submodules

2.7.2 deid.main.identifiers module

deid.main.identifiers.**main**(*args*, *parser*)

2.7.3 deid.main.inspect module

deid.main.inspect.**main**(*args*, *parser*)
Inspect the header fields of dicom files.

inspect currently serves to inspect the header fields of a set of dicom files against a standard, and flag images that don't pass the different levels of criteria

2.7.4 Module contents

deid.main.**get_parser**()

deid.main.**main**()

2.8 deid.utils package

2.8.1 Submodules

2.8.2 deid.utils.actions module

deid.utils.actions.**get_func**(*function_name*)
Get_func will return a function that is defined from a string.

the function is assumed to be in this file

Parameters a function from globals based on a name string(*return*) –

deid.utils.actions.**get_timestamp**(*item_date*, *item_time=None*, *jitter_days=None*, *format=None*)
Get_timestamp will return (default) a UTC timestamp.

This will have some date and (optional) time. A different format can be provided to change default behavior. eg: "%Y%m%d"

deid.utils.actions.**parse_keyvalue_pairs**(*pairs*)
Given a listing of extra arguments, parse into lookup dict.

deid.utils.actions.**parse_value**(*dicom*, *value*, *item=None*, *field=None*, *funcs=None*)
Parse_value will parse the value field of an action.

This function returns either: 1. the string (string or from function) 2. a variable looked up (var:FieldName)

2.8.3 deid.utils.fileio module

deid.utils.fileio.**get_installdir**()
Get installation directory of the application

deid.utils.fileio.**get_temporary_name**(*prefix=None*, *ext=None*)
Get a temporary name.

Get a temporary name, can be used for a directory or file. This does so without creating the file, and adds an optional prefix

Parameters

- **prefix**(*if defined, add the prefix after deid*) –

- **ext** (*if defined, return the file extension appended. Do not specify "."*) –

deid.utils.fileio.**is_number**(*value*)
Determine if the value for a field is numeric

deid.utils.fileio.**read_file**(*filename*, *mode='r'*)
Read a file.

Parameters

- **filename** (*the name of the file to write to*) –
- **mode** (*the mode to open the file, defaults to read (r)*) –

deid.utils.fileio.**read_json**(*filename*, *mode='r'*, *ordered_dict=False*)
Open a file, "filename" and read the string as json

Parameters

- **filename** (*the name of the file to write to*) –
- **mode** (*the mode to open the file, defaults to read (r)*) –
- **ordered_dict** (*If true, return an OrderedDict (default is False)*) –

deid.utils.fileio.**recursive_find**(*base*, *pattern=None*)
Recursively find files that match a pattern.

recursive find will yield dicom files in all directory levels below a base path. It uses get_dcm_files to find the files in the bases.

Parameters

- **base** (*the base directory to search*) –
- **pattern** (*a pattern to match. If None, defaults to "**") –

deid.utils.fileio.**to_int**(*value*)
Convert a value (value) to int, if found to be otherwise

deid.utils.fileio.**write_file**(*filename*, *content*, *mode='w'*)
Write to file.

write_file will open a file, "filename" and write content, "content" and properly close the file

Parameters

- **filename** (*the name of the file to write to*) –
- **content** (*the content to write to file*) –
- **mode** (*the mode to open the file, defaults to write (w)*) –

deid.utils.fileio.**write_json**(*json_obj*, *filename*, *mode='w'*, *print_pretty=True*)
Write a json object to file

Parameters

- **json_obj** (*the dict to print to json*) –
- **filename** (*the output file to write to*) –
- **pretty_print** (*if True, will use nicer formatting*) –

2.8.4 Module contents

PYTHON MODULE INDEX

d

INDEX

E

F

G

H

I