MetaKnowledge Documentation

Release 3.3.2

NetLab

Feb 04, 2019

Contents

A Python3 package for doing computational research on knowledge

metaknowledge is a Python3 package for doing computational research in bibliometrics, scientometrics, and network analysis. It can also be easily used to simplify the process of doing systematic reviews in any disciplinary context.

metaknowledge reads a directory of plain text files containing meta-data on publications and citations, and writes to a variety of data structures that are suitable for longitudinal research, computational text analysis (e.g. topic models and burst analysis), Reference Publication Year Spectroscopy (RPYS), and network analysis (including multi-modal, multi-level, and dynamic). It handles large datasets (e.g. several million records) efficiently.

metaknowledge currently handles data from the Web of Science, PubMed, Scopus, Proquest Dissertations & Theses, and administrative data from the National Science Foundation and the Canadian tri-council granting agencies: SSHRC, CIHR, and NSERC.

Datasets created with metaknowledge can be analyzed using NetworkX and the standard libraries for data analysis in Python. It is also easy to write data to `csv` or `graphml` files for analysis and visualization in R, Stata, Visone, Gephi, or any other tools for data analysis.

metaknowledge also has a simple command line tool for extracting quantitative datasets and network files from Web of Science files. This makes the library more accessible to researchers who do not know Python, and makes it easier to quickly explore new datasets.

CHAPTER 1

Contact

Reid McIlroy-Young, reid@reidmcy.com
University of Chicago, Chicago, IL, USA

John McLevey, john.mclevey@uwaterloo.ca
University of Waterloo, Waterloo, ON, Canada

Jillian Anderson, jillianderson8@gmail.com
University of Waterloo, Waterloo, ON, Canada

CHAPTER 2

Citation

If you are using metaknowledge for research that will be published or publicly distributed, please acknowledge us with the following citation:

> *Reid McIlroy-Young, John McLevey, and Jillian Anderson. 2015. metaknowledge: open source software for social networks, bibliometrics, and sociology of knowledge research. URL: http://www.networkslab.org/metaknowledge.*

Download .bib file:

CHAPTER 3

License

metaknowledge is free and open source software, distributed under the GPL License.

3.1 Installation

Note: For a more recent guide to getting started, please visit the NetLab blog.

metaknowledge has two distributions. The simplest is found under the release branch of the git repo, which can be installed the usual way with pip:

```
pip3 install metaknowledge
```

The second version is at the master branch on *Github*. It comes with extra documents and resources for teaching.

The *download* from *Github* includes a customized *Vagrant* file that installs *metaknowledge* and other useful *Python* libraries into a virtual machine. It is the easiest way of getting *metaknowledge* working if you are not familiar with *Python*.

3.1.1 Install with Vagrant

The *Vagrant* method is intended for students and anyone not familiar with *Python*. It creates a virtual machine with *metaknowledge* installed, as well as the *Python* scientific stack *numpy*, *scipy*, and *matplotlib*, as well as a series of iPython notebooks for teaching *metaknowledge* and *Python*. Some notebooks are more complete than others.

The instructions for those familiar with the command line use the advanced instructions. Otherwise, it is probably best to use the student install.

3.1.2 Student Install

First, you need to install *Vagrant* and *VirtualBox*. You need to do this before you can install *metaknowledge*.

Once *Vagrant* and *VirtualBox* are installed, download *metaknowledge*. Unzip the file. If you are unable to unzip the file, download *7-zip*.

Open the directory *metaknowledge* and go to the vagrant subdirectory. Depending on your operating system, double click either: `win_run`, `mac_run`, or `linux_run`.

A window should pop up and say something like:

```
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Box 'ubuntu/trusty64' could not be found. Attempting to find and install.
↪..
default: Box Provider: virtualbox
default: Box Version: >= 0
```

You will also see an estimate of how long the download and installation process will take (typically 20 minutes). All you have to do it wait for it to finish. When it is done, a browser window will appear at the showing the notebooks. If a browser window opens and it is showing `No data received`, hit refresh a couple times.

When you see a page with the following, you have installed everything successfully:

```
Lesson-1-Getting-Started
Lesson-2-Reading-Files
Lesson-3-Objects
...
```

To open the page again, just double click on which ever of `win_run`, `mac_run`, or `linux_run` you used. It should take less than a minute the second time.

3.1.3 Advanced Instructions

1. Install Vagrant and VirtualBox.
2. Clone the *git repo <https://github.com/networks-lab/metaknowledge.git>*.
3. Make sure you are on the master branch.
4. Go to the vagrant directory.
5. Run `vagrant up`
6. Once vagrant has finished go to http://localhost:1159/

What you are doing by running `vagrant up` is creating an Ubuntu VM and provisioning it with the script `bootstrap`, which is also in the vagrant directory. If you run `vagrant up` again it only starts the VM. To access the VM's notebook once it is created:

1. Go to the vagrant directory.
2. Run `vagrant up`
3. Once vagrant has finished go to http://localhost:1159/

You can also use `vagrant ssh` to ssh into the VM or `vagrant provision` to rerun bootstrap. If `vagrant ssh` does not work on your machine, you should be able to ssh into it at:

```
HostName: 127.0.0.1
Port: 2222
Username: vagrant
Password: vagrant
```

On Windows *PuTTY* has been tested and works well.</p>

3.1.4 Install without Vagrant

Installing without *Vagrant* is done with setuptools. Go to the metaknowledge directory and run `python3 setup.py install`. This is the same version that is installed via `pip` plus some extra development command line tools.

3.1.5 Extending MK

Coming soon

3.1.6 Questions?

If you find bugs, or have questions, please write to:

Reid McIlroy-Young reid@reidmcy.com
John McLevey john.mclevey@uwaterloo.ca

3.1.7 License

3.2 Documentation

3.2.1 Basic Example

metaknoweldge is a Python3 package that simplifies bibliometric and computational analysis of Web of Science data.

To load the data from files and make a network: ::

```
>>> import metaknowledge as mk
>>> RC = mk.RecordCollection("records/")
>>> print(RC)
Collection of 33 records
>>> G = RC.coCiteNetwork(nodeType = 'journal')
Done making a co-citation network of files-from-records                1.1s
>>> print(len(G.nodes()))
223
>>> mk.writeGraph(G, "Cocitation-Network-of-Journals")
```

There is also a simple command line program called `metaknowledge` that comes with the package. It allows for creating networks without any need to know Python. More information about it can be found here.

3.2.2 Overview

This package can read the files downloaded from the Thomson Reuters' Web of Science (*WOS*), Elsevier's Scopus, ProQuest and Medline files from PubMed. These files contain entries on the metadata of scientific records, such as authors, title, and citations. *metaknowledge* can also read grants from various organizations including *NSF* and *NSERC* which are handled similarly to records.

The metaknowledge.RecordCollection class can take a path to one or more of these files load and parse them. The object is the main way for work to be done on multiple records. For each individual record it creates an instance of the metaknowledge.Record class that contains the results of the parsing of the record.

The files read by *metaknowledge* are a databases containing a series of tags (implicitly or explicitly), e.g. `'TI'` is the title for WOS. Each tag has one or more values and metaknowledge can read them and extract useful information. As the tags differ between providers a small set of values can be accessed by special tags, the tags are listed in `commonRecordFields`. These special tags can act on the whole `Record` and as such may contain information provided by any number of other tags.

Citations are handled by a special Citation class. This class can parse the citations given by *WOS* and journals cited by *Scopus* and allows for better comparisons when they are used in graphs.

Note for those reading the docstrings metaknowledge's docs are written in markdown and are processed to produce the documentation found at metaknowledge.readthedocs.io, but you should have no problem reading them from the help function.

3.2.3 Modules

contour

Overview

This is the only module that depends on anything besides *networkx*, it depends on numpy, scipy and matplotlib.

Functions

`metaknowledge.contour.plotting.`**`graphDensityContourPlot`**(*G*, *iters=50*, *layout=None*, *layoutScaleFactor=1*, *overlay=False*, *nodeSize=10*, *axisSamples=100*, *blurringFactor=0.1*, *contours=15*, *graphType='coloured'*)

Creates a 3D plot giving the density of nodes on a 2D plane, as a surface in 3D.

Most of the options are for tweaking the final appearance. *layout* and *layoutScaleFactor* allow a pre-layout graph to be provided. If a layout is not provided the networkx.spring_layout() is used after *iters* iterations. Then, once the graph has been laid out a grid of *axisSamples* cells by *axisSamples* cells is overlaid and the number of nodes in each cell is determined, a gaussian blur is then applied with a sigma of *blurringFactor*. This then forms a surface in 3 dimensions, which is then plotted.

If you find the resultant image looks too banded raise the the *contours* number to ~50.

Parameters

G : `networkx Graph`

The graph to be plotted

iters : `optional [int]`

Default `50`, the number of iterations for the spring layout if *layout* is not provided.

layout : `optional [networkx layout dictionary]`

Default `None`, if provided will be used as a layout of the graph, the maximum distance from the origin along any axis must also given as *layoutScaleFactor*, which is by default `1`.

layoutScaleFactor : `optional [double]`

Default `1`, The maximum distance from the origin allowed along any axis given by *layout*, i.e. the layout must fit in a square centered at the origin with side lengths 2 * *layoutScaleFactor*

overlay : `optional [bool]`

Default `False`, if `True` the 2D graph will be plotted on the X-Y plane at Z = 0.

nodeSize : `optional [double]`

Default `10`, the size of the nodes dawn in the overlay

axisSamples : `optional [int]`

Default 100, the number of cells used along each axis for sampling. A larger number will mean a lower average density.

blurringFactor : `optional [double]`

Default `0.1`, the sigma value used for smoothing the surface density. The higher this number the smoother the surface.

contours : `optional [int]`

Default 15, the number of different heights drawn. If this number is low the resultant image will look very banded. It is recommended this be raised above `50` if you want your images to look good, **Warning** this will make them much slower to generate and interact with.

graphType : `optional [str]`

Default `'coloured'`, if `'coloured'` the image will have a destiny based colourization applied, the only other option is `'solid'` which removes the colourization.

`metaknowledge.contour.plotting.`**`quickVisual`**(*G*, *showLabel=False*)

Just makes a simple *matplotlib* figure and displays it, with each node coloured by its type. You can add labels with *showLabel*. This looks a bit nicer than the one provided my *networkx*'s defaults.

Parameters

showLabel : `optional [bool]`

Default `False`, if `True` labels will be added to the nodes giving their IDs.

grants

Overview

baseGrant

`class` `metaknowledge.grants.baseGrant.`**`FallbackGrant`**(*original*, *grantdDict*, *sFile=''*, *sLine=0*)

A subclass of Grant, it has the same attributes and is returned from the fall back constructor for grants.

`class` `metaknowledge.grants.baseGrant.`**`Grant`**(*original*, *grantdDict*, *idValue*, *bad*, *error*, *sFile=''*, *sLine=0*)

getInstitutions(*tags=None*, *seperator=';'*, *_getTag=False*)

Returns a list of the names of institutions. This is done by looking (in order) for any of fields in *tags* and splitting the strings on *seperator* (in case of multiple institutions). If no strings are found an empty list will be returned.

Note for some Grants `getInstitutions` has been overwritten and will ignore the arguments and simply provide the investigators.

Parameters

tags : `optional list[str]`

A list of the tags to look for institutions in

seperator : `optional str`

The string that separators each institutions name within the column

Returns

`list [str]`

A list of all the found institution's names

getInvestigators(*tags=None*, *seperator=';'*, *_getTag=False*)

Returns a list of the names of investigators. This is done by looking (in order) for any of fields in *tags* and splitting the strings on *seperator*. If no strings are found an empty list will be returned.

Note for some Grants `getInvestigators` has been overwritten and will ignore the arguments and simply provide the investigators.

Parameters

tags : `optional list[str]`

A list of the tags to look for investigators in

seperator : `optional str`

The string that separators each investigators name within the column

Returns

`list [str]`

A list of all the found investigator's names

update(*other*)

Adds all the tag-entry pairs from *other* to the `Grant`. If there is a conflict *other* takes precedence.

Parameters

other : `Grant`

Another `Grant` of the same type as *self*

metaknowledge.grants.baseGrant.**csvAndLinesReader**(*enumeratedFile*, **csvArgs*, ***csvKwargs*)

metaknowledge.grants.baseGrant.**isFallbackGrantFile**(*fileName*, *useFileName=True*, *encoding='latin-1'*, *dialect='excel'*)

metaknowledge.grants.baseGrant.**parserFallbackGrantFile**(*fileName*, *encoding='latin-1'*, *dialect='excel'*)

cihrGrant

class metaknowledge.grants.cihrGrant.**CIHRGrant**(*original*, *grantdDict*, *sFile*, *sLine*)

metaknowledge.grants.cihrGrant.**isCIHRfile**(*fileName*, *useFileName=True*)

metaknowledge.grants.cihrGrant.**parserCIHRfile**(*fileName*)

medlineGrant

class metaknowledge.grants.medlineGrant.**MedlineGrant**(*grantString*)

nsercGrant

class metaknowledge.grants.nsercGrant.**NSERCGrant**(*original*, *grantdDict*, *sFile*, *sLine*)

getInstitutions(*tags=None*, *seperator=';'*, *_getTag=False*)
Returns a list with the names of the institution. The optional arguments are ignored

Returns

`list [str]`

A list with 1 entry the name of the institution

getInvestigators(*tags=None*, *seperator=';'*, *_getTag=False*)
Returns a list of the names of investigators. The optional arguments are ignored.

Returns

`list [str]`

A list of all the found investigator's names

update(*other*)
Adds all the tag-entry pairs from *other* to the `Grant`. If there is a conflict *other* takes precedence.

Parameters

other : `Grant`

Another `Grant` of the same type as *self*

metaknowledge.grants.nsercGrant.**isNSERCfile**(*fileName*, *useFileName=True*)

metaknowledge.grants.nsercGrant.**parserNSERCfile**(*fileName*)

nsfGrant

class metaknowledge.grants.nsfGrant.**NSFGrant**(*grantdDict*, *sFile*)

getInstitutions(*tags=None*, *seperator=';'*, *_getTag=False*)
Returns a list with the names of the institution. The optional arguments are ignored

Returns

list [str]

A list with 1 entry the name of the institution

getInvestigators(*tags=None*, *seperator=';'*, *_getTag=False*)
Returns a list of the names of investigators. The optional arguments are ignored.

Returns

list [str]

A list of all the found investigator's names

metaknowledge.grants.nsfGrant.**isNSFfile**(*fileName*, *useFileName=True*)

metaknowledge.grants.nsfGrant.**parserNSFfile**(*fileName*)

scopusGrant

class metaknowledge.grants.scopusGrant.**ScopusGrant**(*grantString*)

journalAbbreviations

Overview

This module handles the abbreviations, known as J29 abbreviations and given by the J9 tag in WOS Records and for journal titles that WOS employs in citations.

The citations provided by WOS used abbreviated journal titles instead of the full names. The full list of abbreviations can be found at a series pages divided by letter starting at images.webofknowledge.com/WOK46/help/WOS/A_abrvjt.html. The function *updatej9DB()* is used to scape and parse the pages, it must be run without error before the other features can be used. *metaknowledge*. If the database is requested by `getj9dict()`, which is what Citations use, and the database is not found or is corrupted then *updatej9DB()* will be run to download the database if this fails an `mkException` will be raised, the download and parsing usually takes less than a second on a good internet connection.

The other functions of the module are for manually adding and removing abbreviations from the database. It is recommended that this be done with the command-line tool `metaknowledge` instead of with a script.

Functions

metaknowledge.journalAbbreviations.backend.**addToDB**(*abbr=None*, *dbname='manualj9Abbreviations'*)

Adds *abbr* to the database of journals. The database is kept separate from the one scraped from WOS, this supersedes it. The database by default is stored with the WOS one and the name is given by `metaknowledge.journalAbbreviations.manualDBname`. To create an empty database run **addToDB** without an *abbr* argument.

Parameters

abbr : `optional [str or dict[str : str]]`

The journal abbreviation to be added to the database, it can either be a single string in which case that string will be added with its self as the full name, or a dict can be given with the abbreviations as keys and their names as strings, use pipes (`'|'`) to separate multiple names. Note, if the empty string is given as a name the abbreviation will be considered manually **excluded**, i.e. having excludeFromDB() run on it.

dbname : `optional [str]`

The name of the database file, default is `metaknowledge.journalAbbreviations.manualDBname`.

metaknowledge.journalAbbreviations.backend.**excludeFromDB**(*abbr=None*, *dbname='manualj9Abbreviations'*)

Marks *abbr* to be excluded the database of journals. The database is kept separate from the one scraped from WOS, this supersedes it. The database by default is stored with the WOS one and the name is given by `metaknowledge.journalAbbreviations.manualDBname`. To create an empty database run *addToDB()* without an *abbr* argument.

Parameters

abbr : `optional [str or tuple[str] or list[str]`

The journal abbreviation to be excluded from the database, it can either be a single string in which case that string will be exclude or a list/tuple of strings can be given with the abbreviations.

dbname : `optional [str]`

The name of the database file, default is `metaknowledge.journalAbbreviations.manualDBname`.

metaknowledge.journalAbbreviations.backend.**getj9dict**(*dbname='j9Abbreviations'*, *manualDB='manualj9Abbreviations'*, *returnDict='both'*)

Returns the dictionary of journal abbreviations mapping to a list of the associated journal names. By default the local database is used. The database is in the file *dbname* in the same directory as this source file

Parameters

dbname : `optional [str]`

The name of the downloaded database file, the default is determined at run time. It is recommended that this remain untouched.

manualDB : optional [str]

The name of the manually created database file, the default is determined at run time. It is recommended that this remain untouched.

returnDict : optional [str]

default 'both', can be used to get both databases or only one with 'WOS' or 'manual'.

metaknowledge.journalAbbreviations.backend.**j9urlGenerator**(*nameDict=False*)

How to get all the urls for the WOS Journal Title Abbreviations. Each is varies by only a few characters. These are the currently in use urls they may change.

They are of the form:

"https://images.webofknowledge.com/images/help/WOS/%7BVAL%7D_abrvjt.html"
Where {VAL} is a capital letter or the string "0-9"

Returns

list[str]

A list of all the url's strings

metaknowledge.journalAbbreviations.backend.**updatej9DB**(*dbname='j9Abbreviations'*, *saveRawHTML=False*)

Updates the database of Journal Title Abbreviations. Requires an internet connection. The data base is saved relative to the source file not the working directory.

Parameters

dbname : optional [str]

The name of the database file, default is "j9Abbreviations.db"

saveRawHTML : optional [bool]

Determines if the original HTML of the pages is stored, default False. If True they are saved in a directory inside j9Raws begining with todays date.

medline

Overview

These are the functions used to process medline (pubmed) files at the backend. They are meant for use internal use by metaknowledge.

Functions

metaknowledge.medline.medlineHandlers.**isMedlineFile**(*infile*, *checkedLines=2*)

Determines if *infile* is the path to a Medline file. A file is considerd to be a Medline file if it has the correct encoding (latin-1) and within the first *checkedLines* a line starts with "PMID- ".

Parameters

infile : `str`

The path to the targets file

checkedLines : `optional [int]`

default 2, the number of lines to check for the header

Returns

`bool`

`True` if the file is a Medline file

`metaknowledge.medline.medlineHandlers.`**`medlineParser`**(*pubFile*)
Parses a medline file, *pubFile*, to extract the individual entries as *MedlineRecords*.

A medline file is a series of entries, each entry is a series of tags. A tag is a 2 to 4 character string each tag is padded with spaces on the left to make it 4 characters which is followed by a dash and a space (`'- '`). Everything after the tag and on all lines after it not starting with a tag is considered associated with the tag. Each entry's first tag is `PMID`, so a first line looks something like `PMID- 26524502`. Entries end with a single blank line.

Parameters

pubFile : `str`

A path to a valid medline file, use *isMedlineFile* to verify

Returns

`set[MedlineRecord]`

Records for each of the entries

Special Functions

`metaknowledge.medline.tagProcessing.specialFunctions.`**`DOI`**(*R*)

`metaknowledge.medline.tagProcessing.specialFunctions.`**`address`**(*R*)
Gets the first address of the first author

`metaknowledge.medline.tagProcessing.specialFunctions.`**`beginningPage`**(*R*)
As pages may not be given as numbers this is the most accurate this function can be

`metaknowledge.medline.tagProcessing.specialFunctions.`**`month`**(*R*)

`metaknowledge.medline.tagProcessing.specialFunctions.`**`volume`**(*R*)
Returns the first number/word of the volume field, hopefully trimming something like: `'49 Suppl 20'` to `49`

`metaknowledge.medline.tagProcessing.specialFunctions.`**`year`**(*R*)

Tag Functions

metaknowledge.medline.tagProcessing.tagFunctions.**AB**(*val*)

Abstract
basically a one liner after parsing

metaknowledge.medline.tagProcessing.tagFunctions.**AD**(*val*)

Affiliation
Undoing what the parser does then splitting at the semicolons and dropping newlines extra fitlering is required beacuse some AD's end with a semicolon

metaknowledge.medline.tagProcessing.tagFunctions.**AID**(*val*)

ArticleIdentifier
The given values do not require any work

metaknowledge.medline.tagProcessing.tagFunctions.**AU**(*val*)
Author

metaknowledge.medline.tagProcessing.tagFunctions.**AUID**(*val*)

AuthorIdentifier
one line only just need to undo the parser's effects

metaknowledge.medline.tagProcessing.tagFunctions.**BTI**(*val*)
BookTitle

metaknowledge.medline.tagProcessing.tagFunctions.**CI**(*val*)
CopyrightInformation

metaknowledge.medline.tagProcessing.tagFunctions.**CIN**(*val*)
CommentIn

metaknowledge.medline.tagProcessing.tagFunctions.**CN**(*val*)
CorporateAuthor

metaknowledge.medline.tagProcessing.tagFunctions.**CRDT**(*val*)
CreateDate

metaknowledge.medline.tagProcessing.tagFunctions.**CRF**(*val*)
CorrectedRepublishedFrom

metaknowledge.medline.tagProcessing.tagFunctions.**CRI**(*val*)
CorrectedRepublishedIn

metaknowledge.medline.tagProcessing.tagFunctions.**CTI**(*val*)
CollectionTitle

metaknowledge.medline.tagProcessing.tagFunctions.**DA**(*val*)
DateCreated

metaknowledge.medline.tagProcessing.tagFunctions.**DCOM**(*val*)
DateCompleted

metaknowledge.medline.tagProcessing.tagFunctions.**DDIN**(*val*)
DatasetIn

metaknowledge.medline.tagProcessing.tagFunctions.**DEP**(*val*)
DateElectronicPublication

metaknowledge.medline.tagProcessing.tagFunctions.**DP**(*val*)
DatePublication

metaknowledge.medline.tagProcessing.tagFunctions.**DRIN**(*val*)
DatasetUseReportedIn

metaknowledge.medline.tagProcessing.tagFunctions.**EDAT**(*val*)
EntrezDate

metaknowledge.medline.tagProcessing.tagFunctions.**EFR**(*val*)
ErratumFor

metaknowledge.medline.tagProcessing.tagFunctions.**EIN**(*val*)
ErratumIn

metaknowledge.medline.tagProcessing.tagFunctions.**EN**(*val*)
Edition

metaknowledge.medline.tagProcessing.tagFunctions.**FAU**(*val*)
FullAuthor

metaknowledge.medline.tagProcessing.tagFunctions.**FED**(*val*)
Editor

metaknowledge.medline.tagProcessing.tagFunctions.**FIR**(*val*)
InvestigatorFull

metaknowledge.medline.tagProcessing.tagFunctions.**FPS**(*val*)
FullPersonalNameSubject

metaknowledge.medline.tagProcessing.tagFunctions.**GN**(*val*)
GeneralNote

metaknowledge.medline.tagProcessing.tagFunctions.**GR**(*val*)
GrantNumber

metaknowledge.medline.tagProcessing.tagFunctions.**GS**(*val*)
GeneSymbol

metaknowledge.medline.tagProcessing.tagFunctions.**IP**(*val*)
Issue

metaknowledge.medline.tagProcessing.tagFunctions.**IR**(*val*)
Investigator

metaknowledge.medline.tagProcessing.tagFunctions.**IRAD**(*val*)
InvestigatorAffiliation

metaknowledge.medline.tagProcessing.tagFunctions.**IS**(*val*)
ISSN

metaknowledge.medline.tagProcessing.tagFunctions.**ISBN**(*val*)

metaknowledge.medline.tagProcessing.tagFunctions.**JID**(*val*)
NLMID

metaknowledge.medline.tagProcessing.tagFunctions.**JT**(*val*)

JournalTitle

One line only

metaknowledge.medline.tagProcessing.tagFunctions.**LA**(*val*)
Language

metaknowledge.medline.tagProcessing.tagFunctions.**LID**(*val*)
LocationIdentifier

metaknowledge.medline.tagProcessing.tagFunctions.**LR**(*val*)
DateLastRevised

metaknowledge.medline.tagProcessing.tagFunctions.**MH**(*val*)
MeSHTerms

metaknowledge.medline.tagProcessing.tagFunctions.**MHDA**(*val*)
MeSHDate

metaknowledge.medline.tagProcessing.tagFunctions.**MID**(*val*)
ManuscriptIdentifier

metaknowledge.medline.tagProcessing.tagFunctions.**NM**(*val*)
SubstanceName

metaknowledge.medline.tagProcessing.tagFunctions.**OABL**(*val*)
OtherAbstract

metaknowledge.medline.tagProcessing.tagFunctions.**OCI**(*val*)
OtherCopyright

metaknowledge.medline.tagProcessing.tagFunctions.**OID**(*val*)
OtherID

metaknowledge.medline.tagProcessing.tagFunctions.**ORI**(*val*)
OriginalReportIn

metaknowledge.medline.tagProcessing.tagFunctions.**OT**(*val*)

OtherTerm
Nothing needs to be done

metaknowledge.medline.tagProcessing.tagFunctions.**OTO**(*val*)

OtherTermOwner
one line field

metaknowledge.medline.tagProcessing.tagFunctions.**OWN**(*val*)
Owner

metaknowledge.medline.tagProcessing.tagFunctions.**PG**(*val*)

Pagination
all pagination seen so far seems to be only on one line

metaknowledge.medline.tagProcessing.tagFunctions.**PHST**(*val*)
PublicationHistoryStatus

metaknowledge.medline.tagProcessing.tagFunctions.**PL**(*val*)
PlacePublication

metaknowledge.medline.tagProcessing.tagFunctions.**PMC**(*val*)
PubMedCentralIdentifier

metaknowledge.medline.tagProcessing.tagFunctions.**PMCR**(*val*)
PubMedCentralRelease

metaknowledge.medline.tagProcessing.tagFunctions.**PMID**(*val*)
PubMedUniqueIdentifier

metaknowledge.medline.tagProcessing.tagFunctions.**PRIN**(*val*)
PartialRetractionIn

metaknowledge.medline.tagProcessing.tagFunctions.**PROF**(*val*)
PartialRetractionOf

metaknowledge.medline.tagProcessing.tagFunctions.**PS**(*val*)
PersonalNameSubject

metaknowledge.medline.tagProcessing.tagFunctions.**PST**(*val*)
PublicationStatus

metaknowledge.medline.tagProcessing.tagFunctions.**PT**(*val*)
PublicationType

metaknowledge.medline.tagProcessing.tagFunctions.**PUBM**(*val*)
PublishingModel

metaknowledge.medline.tagProcessing.tagFunctions.**RF**(*val*)
NumberReferences

metaknowledge.medline.tagProcessing.tagFunctions.**RIN**(*val*)
RetractionIn

metaknowledge.medline.tagProcessing.tagFunctions.**RN**(*val*)
RegistryNumber

metaknowledge.medline.tagProcessing.tagFunctions.**ROF**(*val*)
RetractionOf

metaknowledge.medline.tagProcessing.tagFunctions.**RPF**(*val*)
RepublishedFrom

metaknowledge.medline.tagProcessing.tagFunctions.**RPI**(*val*)
RepublishedIn

metaknowledge.medline.tagProcessing.tagFunctions.**SB**(*val*)
Subset

metaknowledge.medline.tagProcessing.tagFunctions.**SFM**(*val*)
SpaceFlightMission

metaknowledge.medline.tagProcessing.tagFunctions.**SI**(*val*)
SecondarySourceID

metaknowledge.medline.tagProcessing.tagFunctions.**SO**(*val*)
Source

metaknowledge.medline.tagProcessing.tagFunctions.**SPIN**(*val*)
SummaryForPatients

metaknowledge.medline.tagProcessing.tagFunctions.**STAT**(*val*)
Status

metaknowledge.medline.tagProcessing.tagFunctions.**TA**(*val*)

JournalTitleAbbreviation

One line only

metaknowledge.medline.tagProcessing.tagFunctions.**TI**(*val*)

Title

only one per record

metaknowledge.medline.tagProcessing.tagFunctions.**TT**(*val*)

TransliteratedTitle

metaknowledge.medline.tagProcessing.tagFunctions.**UIN**(*val*)

UpdateIn

metaknowledge.medline.tagProcessing.tagFunctions.**UOF**(*val*)

UpdateOf

metaknowledge.medline.tagProcessing.tagFunctions.**VI**(*val*)

Volume

The volumes as a string as volume is single line

metaknowledge.medline.tagProcessing.tagFunctions.**VTI**(*val*)

VolumeTitle

Backend

class metaknowledge.medline.recordMedline.**MedlineRecord**(*inRecord*, *sFile=''*, *sLine=0*)

Bases: metaknowledge.mkRecord.ExtendedRecord

Class for full Medline(Pubmed) entries.

This class is an ExtendedRecord capable of generating its own id number. You should not create them directly, but instead use medlineParser() on a medline file.

authGenders(*countsOnly=False*, *fractionsMode=False*, *_countsTuple=False*)

Creates a dict mapping `'Male'`, `'Female'` and `'Unknown'` to lists of the names of all the authors.

Parameters

countsOnly : optional bool

Default `False`, if `True` the counts (lengths of the lists) will be given instead of the lists of names

fractionsMode : optional bool

Default `False`, if `True` the fraction counts (lengths of the lists divided by the total number of authors) will be given instead of the lists of names. This supersedes *countsOnly*

Returns

dict[str:str or int]

The mapping of genders to author's names or counts

authors

bibString(*maxLength=1000*, *WOSMode=False*, *restrictedOutput=False*, *niceID=True*)

Makes a string giving the Record as a bibTex entry. If the Record is of a journal article (`PT J`) the bibtext type is set to `'article'`, otherwise it is set to `'misc'`. The ID of the entry is the WOS number and all the Record's fields are given as entries with their long names.

Note This is not meant to be used directly with LaTeX none of the special characters have been escaped and there are a large number of unnecessary fields provided. *niceID* and *maxLength* have been provided to make conversions easier.

Note Record entries that are lists have their values seperated with the string `' and '`

Parameters

maxLength : `optional [int]`

default 1000, The max length for a continuous string. Most bibTex implementation only allow string to be up to 1000 characters (source), this splits them up into substrings then uses the native string concatenation (the `'#'` character) to allow for longer strings

WOSMode : `optional [bool]`

default `False`, if `True` the data produced will be unprocessed and use double curly braces. This is the style WOS produces bib files in and mostly macthes that.

restrictedOutput : `optional [bool]`

default `False`, if `True` the tags output will be limited to tose found in `metaknowledge.commonRecordFields`

niceID : `optional [bool]`

default `True`, if `True` the ID used will be derived from the authors, publishing date and title, if `False` it will be the UT tag

Returns

`str`

The bibTex string of the Record

copy()

Correctly copies the `Record`

Returns

`Record`

A completely decoupled copy of the original

createCitation(*multiCite=False*)

Creates a citation string, using the same format as other WOS citations, for the Record by reading the relevant special tags (`'year'`, `'J9'`, `'volume'`, `'beginningPage'`, `'DOI'`) and using it to create a Citation object.

Parameters

multiCite : `optional [bool]`

Default `False`, if `True` a tuple of Citations is returned with each having a different one of the records authors as the author

Returns

`Citation`

A Citation object containing a citation for the Record.

encoding()
An `abstractmethod`, gives the encoding string of the record.

Returns

`str`

The encoding

get(*tag*, *default=None*, *raw=False*)
Allows access to the raw values or is an Exception safe wrapper to `__getitem__`.

Parameters

tag : `str`

The requested tag

default : `optional [Object]`

Default `None`, the object returned when *tag* is not found

raw : `optional [bool]`

Default `False`, if `True` the unprocessed value of *tag* is returned

Returns

`Object`

The processed value of *tag* or *default*

static getAltName(*tag*)
An `abstractmethod`, gives the alternate name of *tag* or `None`

Parameters

tag : `str`

The requested tag

Returns

`str`

The alternate name of *tag* or `None`

getCitations(*field=None*, *values=None*, *pandasFriendly=True*)

Creates a pandas ready dict with each row a different citation and columns containing the original string, year, journal and author's name.

There are also options to filter the output citations with *field* and *values*

Parameters

field : `optional str`

Default `None`, if given all citations missing the named field will be dropped.

values : `optional str or list[str]`

Default `None`, if *field* is also given only those citations with one of the strings given in *values* will be included.

e.g. to get only citations from 1990 or 1991: `field = year, values = [1991, 1990]`

pandasFriendly : `optional bool`

Default `True`, if `False` a list of the citations will be returned instead of the more complicated pandas dict

Returns

`dict`

A pandas ready dict with all the citations

id

items(*raw=False*)

Like `items` for dicts but with a `raw` option

Parameters

raw : `optional [bool]`

Default `False`, if `True` the `KeysView` contains the raw values as the values

Returns

`KeysView`

The key-value pairs of the record

keys() → a set-like object providing a view on D's keys

sourceFile

sourceLine

specialFuncs(*key*)
An `abstractmethod`, process the special tag, *key* using the whole `Record`

Parameters

key : `str`

One of the special tags: `'authorsFull'`, `'keywords'`, `'grants'`, `'j9'`, `'authorsShort'`, `'volume'`, `'selfCitation'`, `'citations'`, `'address'`, `'abstract'`, `'title'`, `'month'`, `'year'`, `'journal'`, `'beginningPage'` and `'DOI'`

Returns

The processed value of *key*

subDict(*tags*, *raw=False*)
Creates a dict of values of *tags* from the Record. The tags are the keys and the values are the values. If the tag is missing the value will be `None`.

Parameters

tags : `list[str]`

The list of tags requested

raw : `optional [bool]`

default `False` if `True` the retuned values of the dict will be unprocessed

Returns

`dict`

A dictionary with the keys *tags* and the values from the record

static tagProcessingFunc(*tag*)
An `abstractmethod`, gives the function for processing *tag*

Parameters

tag : `optional [str]`

The tag in need of processing

Returns

`function`

The function to process the raw tag

title

values(*raw=False*)
Like `values` for dicts but with a `raw` option

Parameters

raw : `optional [bool]`

Default `False`, if `True` the `ValuesView` contains the raw values

Returns

`ValuesView`

The values of the record

writeRecord(*f*)
This is nearly identical to the original the FAU tag is the only tag not writen in the same place, doing so would require changing the parser and lots of extra logic.

`metaknowledge.medline.recordMedline.`**medlineRecordParser**(*record*)
The parser `MedlineRecord <../classes/MedlineRecord.html#metaknowledge.medline.MedlineRecord>`__ use. This takes an entry from *medlineParser()* and parses it a part of the creation of a `MedlineRecord`.

Parameters

record : `enumerate object`

a file wrapped by `enumerate()`

Returns

`collections.OrderedDict`

An ordered dictionary of the key-vaue pairs in the entry

proquest

Overview

These are the functions used to process medline (pubmed) files at the backend. They are meant for use internal use by metaknowledge.

Functions

`metaknowledge.proquest.proQuestHandlers.`**isProQuestFile**(*infile*, *checkedLines=2*)
Determines if *infile* is the path to a ProQuest file. A file is considered to be a Proquest file if it has the correct encoding (`utf-8`) and within the first *checkedLines* the following starts.

```
Report Information from ProQuest
```

Parameters

infile : `str`

The path to the targets file

checkedLines : `optional [int]`

default 2, the number of lines to check for the header

Returns

`bool`

`True` if the file is a valid ProQuest file

`metaknowledge.proquest.proQuestHandlers.`**`proQuestParser`**(*proFile*)

Parses a ProQuest file, *proFile*, to extract the individual entries.

A ProQuest file has three sections, first a list of the contained entries, second the full metadata and finally a bibtex formatted entry for the record. This parser only uses the first two as the bibtex contains no information the second section does not. Also, the first section is only used to verify the second section. The returned ProQuestRecord contains the data from the second section, with the same key strings as ProQuest uses and the unlabeled sections are called in order, `'Name'`, `'Author'` and `'url'`.

Parameters

proFile : `str`

A path to a valid ProQuest file, use *isProQuestFile* to verify

Returns

`set[ProQuestRecord]`

Records for each of the entries

Special Functions

Tag Functions

`metaknowledge.proquest.tagProcessing.tagFunctions.`**`proQuestClassification`**(*value*)

`metaknowledge.proquest.tagProcessing.tagFunctions.`**`proQuestIdentifier_Keyword`**(*value*)

`metaknowledge.proquest.tagProcessing.tagFunctions.`**`proQuestSubject`**(*value*)

`metaknowledge.proquest.tagProcessing.tagFunctions.`**`proQuestTagToFunc`**(*tag*)

Takes a tag string, *tag*, and returns the processing function for its data. If their is not a predefined function returns the identity function (`lambda x : x`).

Parameters

tag : `str`

The requested tag

Returns

`function`

A function to process the tag's data

Backend

class metaknowledge.proquest.recordProQuest.**ProQuestRecord**(*inRecord*, *recNum=None*, *sFile=''*, *sLine=0*)

Bases: `metaknowledge.mkRecord.ExtendedRecord`

Class for full ProQuest entries.

This class is an ExtendedRecord capable of generating its own id number. You should not create them directly, but instead use proQuestParser() on a ProQuest file.

authGenders(*countsOnly=False*, *fractionsMode=False*, *_countsTuple=False*)

Creates a dict mapping `'Male'`, `'Female'` and `'Unknown'` to lists of the names of all the authors.

Parameters

countsOnly : `optional bool`

Default `False`, if `True` the counts (lengths of the lists) will be given instead of the lists of names

fractionsMode : `optional bool`

Default `False`, if `True` the fraction counts (lengths of the lists divided by the total number of authors) will be given instead of the lists of names. This supersedes *countsOnly*

Returns

`dict[str:str or int]`

The mapping of genders to author's names or counts

authors

bibString(*maxLength=1000*, *WOSMode=False*, *restrictedOutput=False*, *niceID=True*)

Makes a string giving the Record as a bibTex entry. If the Record is of a journal article (`PT J`) the bibtext type is set to `'article'`, otherwise it is set to `'misc'`. The ID of the entry is the WOS number and all the Record's fields are given as entries with their long names.

Note This is not meant to be used directly with LaTeX none of the special characters have been escaped and there are a large number of unnecessary fields provided. *niceID* and *maxLength* have been provided to make conversions easier.

Note Record entries that are lists have their values seperated with the string `' and '`

Parameters

maxLength : `optional [int]`

default 1000, The max length for a continuous string. Most bibTex implementation only allow string to be up to 1000 characters (source), this splits them up into substrings then uses the native string concatenation (the `'#'` character) to allow for longer strings

WOSMode : `optional [bool]`

default `False`, if `True` the data produced will be unprocessed and use double curly braces. This is the style WOS produces bib files in and mostly macthes that.

restrictedOutput : `optional [bool]`

default `False`, if `True` the tags output will be limited to tose found in `metaknowledge.commonRecordFields`

niceID : `optional [bool]`

default `True`, if `True` the ID used will be derived from the authors, publishing date and title, if `False` it will be the UT tag

Returns

`str`

The bibTex string of the Record

`copy``()`
Correctly copies the `Record`

Returns

`Record`

A completely decoupled copy of the original

`createCitation`(*multiCite=False*)
Creates a citation string, using the same format as other WOS citations, for the Record by reading the relevant special tags (`'year'`, `'J9'`, `'volume'`, `'beginningPage'`, `'DOI'`) and using it to create a Citation object.

Parameters

multiCite : `optional [bool]`

Default `False`, if `True` a tuple of Citations is returned with each having a different one of the records authors as the author

Returns

`Citation`

A Citation object containing a citation for the Record.

encoding()
An `abstractmethod`, gives the encoding string of the record.

Returns

`str`

The encoding

get(*tag*, *default=None*, *raw=False*)
Allows access to the raw values or is an Exception safe wrapper to `__getitem__`.

Parameters

tag : `str`

The requested tag

default : `optional [Object]`

Default `None`, the object returned when *tag* is not found

raw : `optional [bool]`

Default `False`, if `True` the unprocessed value of *tag* is returned

Returns

`Object`

The processed value of *tag* or *default*

static getAltName(*tag*)
An `abstractmethod`, gives the alternate name of *tag* or `None`

Parameters

tag : `str`

The requested tag

Returns

`str`

The alternate name of *tag* or `None`

getCitations(*field=None*, *values=None*, *pandasFriendly=True*)
Creates a pandas ready dict with each row a different citation and columns containing the original string, year, journal and author's name.

There are also options to filter the output citations with *field* and *values*

Parameters

field : `optional str`

Default `None`, if given all citations missing the named field will be dropped.

values : `optional str or list[str]`

Default `None`, if *field* is also given only those citations with one of the strings given in *values* will be included.

e.g. to get only citations from 1990 or 1991: `field = year, values = [1991, 1990]`

pandasFriendly : `optional bool`

Default `True`, if `False` a list of the citations will be returned instead of the more complicated pandas dict

Returns

`dict`

A pandas ready dict with all the citations

id

items(*raw=False*)

Like `items` for dicts but with a `raw` option

Parameters

raw : `optional [bool]`

Default `False`, if `True` the `KeysView` contains the raw values as the values

Returns

`KeysView`

The key-value pairs of the record

keys() → a set-like object providing a view on D's keys

sourceFile

sourceLine

specialFuncs(*key*)

An `abstractmethod`, process the special tag, *key* using the whole `Record`

Parameters

key : `str`

One of the special tags: `'authorsFull'`, `'keywords'`, `'grants'`, `'j9'`, `'authorsShort'`, `'volume'`, `'selfCitation'`, `'citations'`, `'address'`, `'abstract'`, `'title'`, `'month'`, `'year'`, `'journal'`, `'beginningPage'` and `'DOI'`

Returns

The processed value of *key*

subDict(*tags*, *raw=False*)
Creates a dict of values of *tags* from the Record. The tags are the keys and the values are the values. If the tag is missing the value will be `None`.

Parameters

tags : `list[str]`

The list of tags requested

raw : `optional [bool]`

default `False` if `True` the retuned values of the dict will be unprocessed

Returns

`dict`

A dictionary with the keys *tags* and the values from the record

static tagProcessingFunc(*tag*)
An `abstractmethod`, gives the function for processing *tag*

Parameters

tag : `optional [str]`

The tag in need of processing

Returns

`function`

The function to process the raw tag

title

values(*raw=False*)
Like `values` for dicts but with a `raw` option

Parameters

raw : `optional [bool]`

Default `False`, if `True` the `ValuesView` contains the raw values

Returns

`ValuesView`

The values of the record

`writeRecord`(*infile*)

An `abstractmethod`, writes the record in its original form to *infile*

Parameters

infile : `writable file`

The file to be written to

`metaknowledge.proquest.recordProQuest.`**`proQuestRecordParser`**(*enRecordFile*, *recNum*)

The parser ProQuestRecords use. This takes an entry from *proQuestParser()* and parses it a part of the creation of a `ProQuestRecord`.

Parameters

enRecordFile : `enumerate object`

a file wrapped by `enumerate()`

recNum : `int`

The number given to the entry in the first section of the ProQuest file

Returns

`collections.OrderedDict`

An ordered dictionary of the key-vaue pairs in the entry

scopus

Overview

Functions

`metaknowledge.scopus.scopusHandlers.`**`isScopusFile`**(*infile*, *checkedLines=2*, *maxHeaderDiff=3*)

Determines if *infile* is the path to a Scopus csv file. A file is considerd to be a Scopus file if it has the correct encoding (`utf-8` with BOM (Byte Order Mark)) and within the first *checkedLines* a line contains the

complete header, the list of all header entries in order is found in `scopus.scopusHeader <#metaknowledge.scopus>‘__.

Note this is for csv files *not* plain text files from scopus, plain text files are not complete.

Parameters

infile : str

The path to the targets file

checkedLines : optional [int]

default 2, the number of lines to check for the header

maxHeaderDiff : optional [int]

default 3, maximum number of different entries in the potetial file from the current known header metaknowledge.scopus.scopusHeader, if exceeded an False will be returned

Returns

bool

True if the file is a Scopus csv file

metaknowledge.scopus.scopusHandlers.**scopusParser**(*scopusFile*)

Parses a scopus file, *scopusFile*, to extract the individual lines as ScopusRecords.

A Scopus file is a csv (Comma-separated values) with a complete header, see `scopus.scopusHeader <#metaknowledge.scopus>‘__ for the entries, and each line after it containing a record’s entry. The string valued entries are quoted with double quotes which means double quotes inside them can cause issues, see *scopusRecordParser()* for more information.

Parameters

scopusFile : str

A path to a valid scopus file, use *isScopusFile()* to verify

Returns

set[ScopusRecord]

Records for each of the entries

Special Functions

Tag Functions

metaknowledge.scopus.tagProcessing.tagFunctions.**citeValue**(*val*)

metaknowledge.scopus.tagProcessing.tagFunctions.**commaSpaceSeperated**(*val*)

metaknowledge.scopus.tagProcessing.tagFunctions.**grantValue**(*val*)

metaknowledge.scopus.tagProcessing.tagFunctions.**integralValue**(*val*)

metaknowledge.scopus.tagProcessing.tagFunctions.**semicolonSeperated**(*val*)

metaknowledge.scopus.tagProcessing.tagFunctions.**semicolonSpaceSeperated**(*val*)

metaknowledge.scopus.tagProcessing.tagFunctions.**stringValue**(*val*)

Backend

class metaknowledge.scopus.recordScopus.**ScopusRecord**(*inRecord*, *sFile=''*, *sLine=0*, *header=None*)

Bases: metaknowledge.mkRecord.ExtendedRecord

Class for full Scopus entries.

This class is an ExtendedRecord capable of generating its own id number. You should not create them directly, but instead use scopusParser() on a scopus **CSV** file.

authGenders(*countsOnly=False*, *fractionsMode=False*, *_countsTuple=False*)

Creates a dict mapping `'Male'`, `'Female'` and `'Unknown'` to lists of the names of all the authors.

Parameters

countsOnly : `optional bool`

Default `False`, if `True` the counts (lengths of the lists) will be given instead of the lists of names

fractionsMode : `optional bool`

Default `False`, if `True` the fraction counts (lengths of the lists divided by the total number of authors) will be given instead of the lists of names. This supersedes *countsOnly*

Returns

`dict[str:str or int]`

The mapping of genders to author's names or counts

authors

bibString(*maxLength=1000*, *WOSMode=False*, *restrictedOutput=False*, *niceID=True*)

Makes a string giving the Record as a bibTex entry. If the Record is of a journal article (`PT J`) the bibtext type is set to `'article'`, otherwise it is set to `'misc'`. The ID of the entry is the WOS number and all the Record's fields are given as entries with their long names.

Note This is not meant to be used directly with LaTeX none of the special characters have been escaped and there are a large number of unnecessary fields provided. *niceID* and *maxLength* have been provided to make conversions easier.

Note Record entries that are lists have their values seperated with the string `' and '`

Parameters

maxLength : `optional [int]`

default 1000, The max length for a continuous string. Most bibTex implementation only allow string to be up to 1000 characters (source), this splits them up into substrings then uses the native string concatenation (the `'#'` character) to allow for longer strings

WOSMode : `optional [bool]`

default `False`, if `True` the data produced will be unprocessed and use double curly braces. This is the style WOS produces bib files in and mostly macthes that.

restrictedOutput : `optional [bool]`

default `False`, if `True` the tags output will be limited to tose found in `metaknowledge.commonRecordFields`

niceID : `optional [bool]`

default `True`, if `True` the ID used will be derived from the authors, publishing date and title, if `False` it will be the UT tag

Returns

`str`

The bibTex string of the Record

copy()
Correctly copies the `Record`

Returns

`Record`

A completely decoupled copy of the original

createCitation(*multiCite=False*)
Overwriting the general citation creator to deal with scopus weirdness.

Creates a citation string, using the same format as other WOS citations, for the Record by reading the relevant special tags (`'year'`, `'J9'`, `'volume'`, `'beginningPage'`, `'DOI'`) and using it to create a Citation object.

Parameters

multiCite : `optional [bool]`

Default `False`, if `True` a tuple of Citations is returned with each having a different one of the records authors as the author

Returns

`Citation`

A Citation object containing a citation for the Record.

encoding()
An `abstractmethod`, gives the encoding string of the record.

Returns

`str`

The encoding

get (*tag*, *default=None*, *raw=False*)
Allows access to the raw values or is an Exception safe wrapper to `__getitem__`.

Parameters

tag : `str`

The requested tag

default : `optional [Object]`

Default `None`, the object returned when *tag* is not found

raw : `optional [bool]`

Default `False`, if `True` the unprocessed value of *tag* is returned

Returns

`Object`

The processed value of *tag* or *default*

static getAltName (*tag*)
An `abstractmethod`, gives the alternate name of *tag* or `None`

Parameters

tag : `str`

The requested tag

Returns

`str`

The alternate name of *tag* or `None`

getCitations (*field=None*, *values=None*, *pandasFriendly=True*)
Creates a pandas ready dict with each row a different citation and columns containing the original string, year, journal and author's name.

There are also options to filter the output citations with *field* and *values*

Parameters

field : `optional str`

Default `None`, if given all citations missing the named field will be dropped.

values : optional str or list[str]

Default None, if *field* is also given only those citations with one of the strings given in *values* will be included.

e.g. to get only citations from 1990 or 1991: field = year, values = [1991, 1990]

pandasFriendly : optional bool

Default True, if False a list of the citations will be returned instead of the more complicated pandas dict

Returns

dict

A pandas ready dict with all the citations

id

items(*raw=False*)

Like items for dicts but with a raw option

Parameters

raw : optional [bool]

Default False, if True the KeysView contains the raw values as the values

Returns

KeysView

The key-value pairs of the record

keys() → a set-like object providing a view on D's keys

sourceFile

sourceLine

specialFuncs(*key*)

An abstractmethod, process the special tag, *key* using the whole Record

Parameters

key : str

One of the special tags: 'authorsFull', 'keywords', 'grants', 'j9', 'authorsShort', 'volume', 'selfCitation', 'citations', 'address', 'abstract', 'title', 'month', 'year', 'journal', 'beginningPage' and 'DOI'

Returns

The processed value of *key*

subDict(*tags*, *raw=False*)
Creates a dict of values of *tags* from the Record. The tags are the keys and the values are the values. If the tag is missing the value will be `None`.

Parameters

tags : `list[str]`

The list of tags requested

raw : `optional [bool]`

default `False` if `True` the retuned values of the dict will be unprocessed

Returns

`dict`

A dictionary with the keys *tags* and the values from the record

static tagProcessingFunc(*tag*)
An `abstractmethod`, gives the function for processing *tag*

Parameters

tag : `optional [str]`

The tag in need of processing

Returns

`function`

The function to process the raw tag

title

values(*raw=False*)
Like `values` for dicts but with a `raw` option

Parameters

raw : `optional [bool]`

Default `False`, if `True` the `ValuesView` contains the raw values

Returns

`ValuesView`

The values of the record

writeRecord(*f*)
An `abstractmethod`, writes the record in its original form to *infile*

Parameters

infile : `writable file`

The file to be written to

`metaknowledge.scopus.recordScopus.`**scopusRecordParser**(*record*, *header=None*)
The parser ScopusRecords use. This takes a line from *scopusParser()* and parses it as a part of the creation of a `ScopusRecord`.

Note this is for csv files downloaded from scopus *not* the text records as those are less complete. Also, Scopus uses double quotes (`"`) to quote strings, such as abstracts, in the csv so double quotes in the string must be escaped. For reasons not fully understandable by mortals they choose to use two double quotes in a row (`""`) to represent an escaped double quote. This parser does not unescape these quotes, but it does correctly handle their interacts with the outer double quotes.

Parameters

record : `str`

string ending with a newline containing the record's entry

Returns

`dict`

A dictionary of the key-vaue pairs in the entry

WOS

Overview

These are the functions used to process medline (pubmed) files at the backend. They are meant for use internal use by metaknowledge.

Functions

`metaknowledge.WOS.wosHandlers.`**isWOSFile**(*infile*, *checkedLines=3*)
Determines if *infile* is the path to a WOS file. A file is considerd to be a WOS file if it has the correct encoding (`utf-8` with a BOM) and within the first *checkedLines* a line starts with `"VR 1.0"`.

Parameters

infile : `str`

The path to the targets file

checkedLines : `optional [int]`

default 2, the number of lines to check for the header

Returns

`bool`

`True` if the file is a WOS file

`metaknowledge.WOS.wosHandlers.`**`wosParser`**`(`*`isifile`*`)`

This is a function that is used to create RecordCollections from files.

wosParser() reads the file given by the path isifile, checks that the header is correct then reads until it reaches EF. All WOS records it encounters are parsed with *recordParser()* and converted into Records. A list of these `Records` is returned.

`BadWOSFile` is raised if an issue is found with the file.

Parameters

isifile : `str`

The path to the target file

Returns

`List[Record]`

All the `Records` found in *isifile*

Help Functions

`metaknowledge.WOS.tagProcessing.helpFuncs.`**`getMonth`**`(`*`s`*`)`

Known formats:
Month ("%b")
Month Day ("%b %d")
Month-Month ("%b-%b") — this gets coerced to the first %b, dropping the month range
Season ("%s") — this gets coerced to use the first month of the given season
Month Day Year ("%b %d %Y")
Month Year ("%b %Y")
Year Month Day ("%Y %m %d")

`metaknowledge.WOS.tagProcessing.helpFuncs.`**`makeBiDirectional`**`(`*`d`*`)`

Helper for generating tagNameConverter

Makes dict that maps from key to value and back

metaknowledge.WOS.tagProcessing.helpFuncs.**reverseDict**(*d*)

Helper for generating fullToTag
Makes dict of value to key

Tag Functions

metaknowledge.WOS.tagProcessing.tagFunctions.**DOI**(*val*)

The DI Tag

return the DOI number of the record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The DOI number string

metaknowledge.WOS.tagProcessing.tagFunctions.**ISBN**(*val*)

The BN Tag

extracts a list of ISBNs associated with the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list`

The ISBNs

metaknowledge.WOS.tagProcessing.tagFunctions.**ISSN**(*val*)

The SN Tag

extracts the ISSN of the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The ISSN string

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`ResearcherIDnumber`**`(`*`val`*`)`

The RI Tag

extracts a list of the research IDs of the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list[str]`

The list of the research IDs

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`abstract`**`(`*`val`*`)`

The AB Tag

return abstract of the record, with newlines hopefully in the correct places

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The abstract

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`articleNumber`**`(`*`val`*`)`

The AR Tag

extracts a string giving the article number, not all are integers

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The article number

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`authAddress`**`(`*`val`*`)`

The C1 Tag

extracts the address of the authors as given by WOS. **Warning** the mapping of author to address is not very good and is given in multiple ways.

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list[str]`

A list of addresses

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`authKeywords`**`(`*`val`*`)`

The DE Tag

extracts the keywords assigned by the author of the Record. The WOS description is:

```
Author keywords are included in records of articles from 1991 forward. They are
↪also include in conference proceedings records.
```

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list[str]`

The list of keywords

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`authorsFull`**`(`*`val`*`)`

The AF Tag

extracts a list of authors full names

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list[str]`

A list of author's names

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`authorsShort`**`(`*`val`*`)`

The AU Tag

extracts a list of authors shortened names

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list[str]`

A list of shortened author's names

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`beginningPage`**`(`*`val`*`)`

The BP Tag

extracts the first page the record occurs on, not all are integers

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The first page number

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`bookAuthor`**`(`*`val`*`)`

The BA Tag

extracts a list of the short names of the authors of a book Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list[str]`

A list of shortened author's names

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`bookAuthorFull`**`(`*`val`*`)`

The BF Tag

extracts a list of the long names of the authors of a book Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list[str]`

A list of author's names

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`bookDOI`**`(`*`val`*`)`

The D2 Tag

extracts the book DOI of the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The DOI number

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`citations`**`(`*`val`*`)`

The CR Tag

extracts a list of all the citations in the record, the citations are the metaknowledge.Citation class.

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list[metaknowledge.Citation]`

A list of Citations

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`citedRefsCount`**`(`*`val`*`)`

The NR Tag

extracts the number citations, length of CR list

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`int`

The number of CRs

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`confDate`**`(`*`val`*`)`

The CY Tag

extracts the date string of the conference associated with the Record, the date is not normalized

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The data of the conference

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`confHost`**`(`*`val`*`)`

The HO Tag

extracts the host of the conference

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The host

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`confLocation`**`(`*`val`*`)`

The CL Tag

extracts the sting giving the conference's location

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The conferences address

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`confSponsors`**`(`*`val`*`)`

The SP Tag

extracts a list of sponsors for the conference associated with the record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

A the list of of sponsors

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`confTitle`**`(`*`val`*`)`

The CT Tag

extracts the title of the conference associated with the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The title of the conference

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`docType`**`(`*`val`*`)`

The DT Tag

extracts the type of document the Record contains

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The type of the Record

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`documentDeliveryNumber`**`(`*`val`*`)`

The GA Tag

extracts the document delivery number of the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The document delivery number

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`eISSN`**`(`*`val`*`)`

The EI Tag

extracts the EISSN of the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The EISSN string

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`editedBy`**`(`*`val`*`)`

The BE Tag

extracts a list of the editors of the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list[str]`

A list of editors

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`editors`**`(`*`val`*`)`

Needs Work

currently not well understood, returns *val*

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`email`**`(`*`val`*`)`

The EM Tag

extracts a list of emails given by the authors of the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list[str]`

A list of emails

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`endingPage`**`(`*`val`*`)`

The EP Tag

return the last page the record occurs on as a string, not aall are intergers

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The final page number

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`funding`**`(`*`val`*`)`

The FU Tag

extracts a list of the groups funding the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list[str]`

A list of funding groups

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`fundingText`**`(`*`val`*`)`

The FX Tag

extracts a string of the funding thanks

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The funding thank-you

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`group`**`(`*`val`*`)`

The GP Tag

extracts the group associated with the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

A the name of the group

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`groupName`**`(`*`val`*`)`

The CA Tag

extracts the name of the group associated with the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The group's name

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`isoAbbreviation`**`(`*`val`*`)`

The JI Tag

extracts the iso abbreviation of the journal

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The iso abbreviation of the journal

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`issue`**`(`*`val`*`)`

The IS Tag

extracts a string giving the issue or range of issues the Record was in, not all are integers

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The issue number/range

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`j9`**`(`*`val`*`)`

The J9 Tag

extracts the J9 (29-Character Source Abbreviation) of the publication

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The 29-Character Source Abbreviation

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`journal`**(*val*)

The SO Tag

extracts the full name of the publication and normalizes it to uppercase

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The name of the journal

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`keywords`**(*val*)

The ID Tag

extracts the WOS keywords of the Record. The WOS description is:

```
KeyWords Plus are index terms created by Thomson Reuters from significant,␣
↪frequently occurring words in the titles of an article's cited references.
```

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list[str]`

The keyWords list

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`language`**`(`*`val`*`)`

The LA Tag

extracts the languages of the Record as a string with languages separated by ‘, ‘, usually there is only one language

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The language(s) of the record

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`meetingAbstract`**`(`*`val`*`)`

The MA Tag

extracts the ID of the meeting abstract prefixed by ‘EPA-‘

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The meeting abstract prefixed

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`month`**`(`*`val`*`)`

The PD Tag

extracts the month the record was published in as an int with January as 1, February 2, . . .

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`int`

A integer giving the month

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`orcID`**`(`*`val`*`)`

The OI Tag

extracts a list of orc IDs of the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The orc ID

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`pageCount`**`(`*`val`*`)`

The PG Tag

returns an integer giving the number of pages of the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`int`

The page count

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`partNumber`**`(`*`val`*`)`

The PN Tag

return an integer giving the part of the issue the Record is in

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`int`

The part of the issue of the Record

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`pubMedID`**`(`*`val`*`)`

The PM Tag

extracts the pubmed ID of the record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The pubmed ID

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`pubType`**`(`*`val`*`)`

The PT Tag

extracts the type of publication as a character: conference, book, journal, book in series, or patent

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

A string

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`publisher`**`(`*`val`*`)`

The PU Tag

extracts the publisher of the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The publisher

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`publisherAddress`**`(`*`val`*`)`

The PA Tag

extracts the publishers address

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The publisher address

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`publisherCity`**`(`*`val`*`)`

The PI Tag

extracts the city the publisher is in

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The city of the publisher

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`reprintAddress`**`(`*`val`*`)`

The RP Tag

extracts the reprint address string

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The reprint address

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`seriesSubtitle`**`(`*`val`*`)`

The BS Tag

extracts the title of the series the Record is in

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The subtitle of the series

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`seriesTitle`**`(`*`val`*`)`

The SE Tag

extracts the title of the series the Record is in

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The title of the series

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`specialIssue`**`(`*`val`*`)`

The SI Tag

extracts the special issue value

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The special issue value

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`subjectCategory`**`(`*`val`*`)`

The SC Tag

extracts a list of the subjects associated with the Record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list[str]`

A list of the subjects associated with the Record

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`subjects`**`(`*`val`*`)`

The WC Tag

extracts a list of subjects as assigned by WOS

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`list[str]`

The subjects list

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`supplement`**`(`*`val`*`)`

The SU Tag

extracts the supplement number

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The supplement number

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`title`**`(`*`val`*`)`

The TI Tag

extracts the title of the record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The title of the record

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`totalTimesCited`**`(`*`val`*`)`

The Z9 Tag

extracts the total number of citations of the record

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`int`

The total number of citations

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`volume`**`(`*`val`*`)`

The VL Tag

return the volume the record is in as a string, not all are integers

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The volume number

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`wosString`**`(`*`val`*`)`

The UT Tag

extracts the WOS number of the record as a string preceded by "WOS:"

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`str`

The WOS number

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`wosTimesCited`**`(`*`val`*`)`

The TC Tag

extracts the number of times the Record has been cited by records in WOS

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`int`

The number of time the Record has been cited

`metaknowledge.WOS.tagProcessing.tagFunctions.`**`year`**`(`*`val`*`)`

The PY Tag

extracts the year the record was published in as an int

Parameters

val: `list[str]`

The raw data from a WOS file

Returns

`int`

The year

Dict Functions

`metaknowledge.WOS.tagProcessing.funcDicts.`**`isTagOrName`**(*val*)
Checks if *val* is a tag or full name of tag if so returns `True`

Parameters

val: `str`

A string possible forming a tag or name

Returns

`bool`

`True` if *val* is a tag or name, otherwise `False`

`metaknowledge.WOS.tagProcessing.funcDicts.`**`normalizeToName`**(*val*)
Converts tags or full names to full names, case sensitive

Parameters

val: `str`

A two character string giving the tag or its full name

Returns

`str`

The full name of *val*

`metaknowledge.WOS.tagProcessing.funcDicts.`**`normalizeToTag`**(*val*)
Converts tags or full names to 2 character tags, case insensitive

Parameters

val: `str`

A two character string giving the tag or its full name

Returns

`str`

The short name of *val*

`metaknowledge.WOS.tagProcessing.funcDicts.`**`tagToFull`**`(`*`tag`*`)`
A wrapper for `tagToFullDict`, it maps 2 character tags to their full names.

Parameters

tag: `str`

A two character string giving the tag

Returns

`str`

The full name of *tag*

Backend

This file contains the Record class for metaknowledge and one helper function for parsing WOS records, recordParser. The record class is used to represent a single records meta-data from WOS.

`class` `metaknowledge.WOS.recordWOS.`**`WOSRecord`**`(`*`inRecord`*, *`sFile=''`*, *`sLine=0`*`)`
Bases: `metaknowledge.mkRecord.ExtendedRecord`

Class for full WOS records

It is meant to be immutable; many of the methods and attributes are evaluated when first called, not when the object is created, and the results are stored privately.

The record's meta-data is stored in an ordered dictionary labeled by WOS tags. To access the raw data stored in the original record the tags() method can be used. To access data that has been processed and cleaned the attributes named after the tags are used.

Customizations

The `Record`'s hashing and equality testing are based on the WOS number (the tag is 'UT', and also called the accession number). They are strings starting with `'WOS:'` and followed by 15 or so numbers and letters, although both the length and character set are known to vary. The numbers are unique to each record so are used for comparisons. If a record is `bad` all equality checks return `False`.

When converted to a string the records title is used so for a record `R`, `R.TI == R.title == str(R)` and its representation uses the WOS number instead of memory location.

Attributes

When a record is created if the parsing of the WOS file failed it is marked as `bad`. The `bad` attribute is set to True and the `error` attribute is created to contain the exception object.

Generally, to get the information from a Record its attributes should be used. For a Record `R`, calling `R.CR` causes citations() from the the tagProcessing module to be called on the contents of the raw 'CR' field. Then the result is saved and returned. In this case, a list of Citation objects is returned. You can also call `R.citations` to get the same effect, as each known field tag has a longer name (currently there are 61 field tags). These names are meant to make accessing tags more readable and mapping from tag to name can be found in the tagToFull dict. If a tag is known (in tagToFull) but not in the raw data `None` is returned instead. Most tags when cleaned return a string or list of strings, the exact results can be found in the help for the particular function.

The attribute `authors` is also defined as a convenience and returns the same as 'AF' or if that is not found 'AU'.

__Init__

Records are generally created as collections in Recordcollections, and not as individual objects. If you wish to create one on its own it is possible, the arguments are as follows.

Parameters

inRecord: `files stream, dict, str or itertools.chain`

If it is a file stream the file must be open at the location of the first tag in the record, usually 'PT', and the file will be read until 'ER' is found, which indicates the end of the record in the file.

If a dict is passed the dictionary is used as the database of fields and tags, so each key is considered a WOS tag and each value a list of the lines of the original associated with the tag. This is the same form of dict that recordParser returns.

For a string the input must be the raw textual data of a single record in the WOS style, like the file stream it must start at the first tag and end in `'ER'`.

itertools.chain is treated identically to a file stream and is used by RecordCollections.

sFile : `optional [str]`

Is the name of the file the raw data was in, by default it is blank. It is mostly used to make error messages more informative.

sLine : `optional [int]`

Is the line the record starts on in the raw data file. It is mostly used to make error messages more informative.

UT

Returns the UT tag (WOS number) of the record

authGenders(*countsOnly=False*, *fractionsMode=False*, *_countsTuple=False*)

Creates a dict mapping `'Male'`, `'Female'` and `'Unknown'` to lists of the names of all the authors.

authors

bibString(*maxLength=1000*, *WOSMode=False*, *restrictedOutput=False*, *niceID=True*)

Makes a string giving the Record as a bibTex entry. If the Record is of a journal article (`PT J`) the bibtext type is set to `'article'`, otherwise it is set to `'misc'`. The ID of the entry is the WOS number and all the Record's fields are given as entries with their long names.

Note This is not meant to be used directly with LaTeX none of the special characters have been escaped and there are a large number of unnecessary fields provided. *niceID* and *maxLength* have been provided to make conversions easier.

Note Record entries that are lists have their values seperated with the string `' and '`

copy()
Correctly copies the `Record`

createCitation(*multiCite=False*)
Creates a citation string, using the same format as other WOS citations, for the Record by reading the relevant special tags (`'year'`, `'J9'`, `'volume'`, `'beginningPage'`, `'DOI'`) and using it to create a Citation object.

encoding()
An `abstractmethod`, gives the encoding string of the record.

get(*tag*, *default=None*, *raw=False*)
Allows access to the raw values or is an Exception safe wrapper to `__getitem__`.

static getAltName(*tag*)
An `abstractmethod`, gives the alternate name of *tag* or `None`

getCitations(*field=None*, *values=None*, *pandasFriendly=True*)
Creates a pandas ready dict with each row a different citation and columns containing the original string, year, journal and author's name.

There are also options to filter the output citations with *field* and *values*

id

items(*raw=False*)
Like `items` for dicts but with a `raw` option

keys() → a set-like object providing a view on D's keys

sourceFile

sourceLine

specialFuncs(*key*)
An `abstractmethod`, process the special tag, *key* using the whole `Record`

subDict(*tags*, *raw=False*)
Creates a dict of values of *tags* from the Record. The tags are the keys and the values are the values. If the tag is missing the value will be `None`.

static tagProcessingFunc(*tag*)
An `abstractmethod`, gives the function for processing *tag*

title

values(*raw=False*)
Like `values` for dicts but with a `raw` option

wosString
Returns the WOS number (UT tag) of the record

writeRecord(*infile*)
Writes to *infile* the original contents of the Record. This is intended for use by RecordCollections to write to file. What is written to *infile* is bit for bit identical to the original record file (if utf-8 is used). No newline is inserted above the write but the last character is a newline.

`metaknowledge.WOS.recordWOS.`**recordParser**(*paper*)
This is function that is used to create Records from files.

recordParser() reads the file *paper* until it reaches 'ER'. For each field tag it adds an entry to the returned dict with the tag as the key and a list of the entries as the value, the list has each line separately, so for the following two lines in a record:

```
AF BREVIK, I
   ANICIN, B
```

The entry in the returned dict would be `{'AF' : ["BREVIK, I", "ANICIN, B"]}`

`Record` objects can be created with these dictionaries as the initializer.

Parameters

paper : `file stream`

An open file, with the current line at the beginning of the WOS record.

Returns

`OrderedDict[str : List[str]]`

A dictionary mapping WOS tags to lists, the lists are of strings, each string is a line of the record associated with the tag.

3.2.4 Classes

CIHRGrant(Grant)

class metaknowledge.grants.cihrGrant.**CIHRGrant**(*original*, *grantdDict*, *sFile*, *sLine*)

metaknowledge.grants.cihrGrant.**isCIHRfile**(*fileName*, *useFileName=True*)

metaknowledge.grants.cihrGrant.**parserCIHRfile**(*fileName*)

Citation(Hashable)

class metaknowledge.citation.**Citation**(*cite*, *scopusMode=False*)

A class to hold citation strings and allow for comparison between them.

The initializer takes in a string representing a WOS citation in the form:

```
Author, Year, Journal, Volume, Page, DOI
```

`Author` is the author's name in the form of first last name first initial sometimes followed by a period.
`Year` is the year of publication.
`Journal` being the 29-Character Source Abbreviation of the journal.
`Volume` is the volume number(s) of the publication preceded by a V
`Page` is the page number the record starts on
`DOI` is the DOI number of the cited record preceeded by the letters `'DOI'`
Combined they look like:

```
Nunez R., 1998, MATH COGNITION, V4, P85, DOI 10.1080/135467998387343
```

Note: any of the fields have been known to be missing and the requirements for the fields are not always met. If something is in the source string that cannot be interpreted as any of these it is put in the `misc` attribute. That is the reason to use this class, it gracefully handles missing information while still allowing for comparison between WOS citation strings.

Customizations

Citation's hashing and equality checking are based on *ID()* and use the values of `author`, `year` and `journal`.

When converted to a string a Citation will return the original string.

Attributes

As noted above, citations are considered to be divided into six distinct fields (`Author`, `Year`, `Journal`, `Volume`, `Page` and `DOI`) with a seventh `misc` for anything not in those. Records thus have an attribute with a name corresponding to each `author`, `year`, `journal`, `V`, `P`, `DOI` and `misc` respectively. These are created if there is anything in the field. So a `Citation` created from the string: `'Nunez R., 1998, MATH COGNITION'` would have `author`, `year` and `journal` defined. While one from `'Nunez R.'` would have only the attribute `misc`.

If the parsing of a citation string fails the attribute `bad` is set to `True` and the attribute `error` is created to contain said error, which is a BadCitation object. If no errors occur `bad` is `False`.

The attribute `original` is the unmodified string (*cite*) given to create the Citation, it can also be accessed by converting to a string, e.g. with `str()`.

__Init__

Citations can be created by Records or by giving the initializer a string containing a WOS style citation.

Parameters

cite : `str`

A str containing a WOS style citation.

`Extra``()`
Returns any `V`, `P`, `DOI` or `misc` values as a string. These are all the values not returned by *ID()*, they are separated by `' ,'`.

`FullJournalName``()`
Returns the full name of the Citation's journal field. Requires the j9Abbreviations database file.

Note: Requires the j9Abbreviations database file and will raise an error if it cannot be found.

`ID``()`
Returns all of `author`, `year` and `journal` available separated by `' ,'`. It is for shortening labels when creating networks as the resultant strings are often unique. *Extra()* gets everything not returned by **ID**().

This is also used for hashing and equality checking.

__eq__(*other*)
First checks DOI for equality then checks each attribute if any are not equal False is returned

__hash__()
A hash for Citation that should be equal to the hash of other citations that are equal to it. Based on the values returned by *ID()*.

__init__(*cite*, *scopusMode=False*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
the representation of the Citation is its original form

__str__()
returns the original string

__weakref__
list of weak references to the object (if defined)

addToDB(*manualName=None*, *manualDB='manualj9Abbreviations'*, *invert=False*)
Adds the journal of this Citation to the user created database of journals. This will cause *isJournal()* to return `True` for this Citation and all others with its `journal`.

Note: Requires the j9Abbreviations database file and will raise an error if it cannot be found.

allButDOI()
Returns a string of the normalized values from the Citation excluding the DOI number. Equivalent to getting the ID with *ID()* then appending the extra values from *Extra()* and then removing the substring containing the DOI number.

isAnonymous()
Checks if the author is given as `'[ANONYMOUS]'` and returns `True` if so.

isJournal(*dbname='j9Abbreviations'*, *manualDB='manualj9Abbreviations'*, *returnDict='both'*, *checkIfExcluded=False*)
Returns `True` if the `Citation`'s `journal` field is a journal abbreviation from the WOS listing found at http://images.webofknowledge.com/WOK46/help/WOS/A_abrvjt.html, i.e. checks if the citation is citing a journal.

Note: Requires the j9Abbreviations database file and will raise an error if it cannot be found.

Note: All parameters are used for getting the data base with getj9dict.

metaknowledge.citation.**filterNonJournals**(*citesLst*, *invert=False*)
Removes the `Citations` from *citesLst* that are not journals

Parameters

citesLst : `list [Citation]`

A list of citations to be filtered

invert : `optional [bool]`

Default `False`, if `True` non-journals will be kept instead of journals

Returns

`list [Citation]`

A filtered list of Citations from *citesLst*

Collection(MutableSet, Hashable)

class metaknowledge.**Collection**(*inSet*, *allowedTypes*, *collectedTypes*, *name*, *bad*, *errors*, *quietStart=False*)

A named hashable set with some error reporting.

`Collections` have all the methods of builtin `sets` as well as error reporting with *bad* and *error*, and control over the contained items with *allowedTypes* and *collectedTypes*.

Customizations

When created *name* should be a string that allows users to easily determine the source of the `Collection`

When created the you must provided a set of types, *allowedTypes*, when new items are added they will be checked and if they are not instances of any of the types an `CollectionTypeError` exception will be raised. The *collectedTypes* set that is provided should be a set of only the types in the `Collection`.

If any of the elements in the `Collection` are bad then *bad* should be set to `True` and the `dict` *errors* should map the item to it's exception.

All of these customizations are managed when operations occur on the `Collection` and if 2 `Collections` are modified with one of the binary operators (`|`, `-`, etc) the `_collectedTypes` and `errors` attributes will be modified the same way. `name` will be updated to explain the operation(s) that occurred.

__Init__

As `Collection` is mostly meant to be base for other classes all but one of the arguments in the __Init__ are not optional and the optional one is not used.

Parameters

inSet : `set`

The objects to be contained

allowedTypes : `set[type]`

A set of types, `{object}` will allow virtually everything

collectedTypes : `set[type]`

The types (or supertypes) of the objects in *inSet*

name : `str`

The name of the `Collection`

bad : `bool`

If any of the elements are bad

errors : `dict[:Exception]`

A mapping from items to their errors

quietStart : `optional [bool]`

Default `False`, does nothing. This is here for use as a interface by subclasses

__eq__(*other*)

Return self==value.

__ge__(*other*)
Return self>=value.

__hash__()
Return hash(self).

__init__(*inSet*, *allowedTypes*, *collectedTypes*, *name*, *bad*, *errors*, *quietStart=False*)
Basically a collections.abc.MutableSet wrapper for a set with a bunch of extra record keeping attached.

__le__(*other*)
Return self<=value.

__repr__()
Return repr(self).

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

add(*elem*)
Adds *elem* to the collection.

chunk(*maxSize*)
Splits the `Collection` into *maxSize* size or smaller `Collections`

clear()
"Removes all elements from the collection and resets the error handling

copy()
Creates a shallow copy of the collection

discard(*elem*)
Removes *elem* from the collection, will not raise an Exception if *elem* is missing

peek()
returns a random element from the collection. If ran twice the same element will usually be returned

pop()
Removes a random element from the collection and returns it

remove(*elem*)
Removes *elem* from the collection, will raise a KeyError is *elem* is missing

split(*maxSize*)
Destructively, splits the `Collection` into *maxSize* size or smaller `Collections`. The source `Collection` will be empty after this operation

CollectionWithIDs(Collection)

class metaknowledge.**CollectionWithIDs**(*inSet*, *allowedTypes*, *collectedTypes*, *name*, *bad*, *errors*, *quietStart=False*)
A Collection with a few extra methods that assume all the contained items have an id attribute and a bad attribute, e.g. Records or Grants.

__Init__

As `CollectionWithIDs` is mostly meant to be base for other classes all but one of the arguments in the `__init__` are not optional and the optional one is not used. The `__init__()` function is the same as a Collection.

__init__(*inSet*, *allowedTypes*, *collectedTypes*, *name*, *bad*, *errors*, *quietStart=False*)
Basically a collections.abc.MutableSet wrapper for a set with a bunch of extra record keeping attached.

badEntries()
Creates a new collection of the same type with only the bad entries

Returns

`CollectionWithIDs`

A collection of only the bad entries

containsID(*idVal*)
Checks if the collected items contains the give *idVal*

Parameters

idVal : `str`

The queried id string

Returns

`bool`

`True` if the item is in the collection

cooccurrenceCounts(*keyTag*, **countedTags*)
Counts the number of times values from any of the *countedTags* occurs with *keyTag*. The counts are retuned as a dictionary with the values of *keyTag* mapping to dictionaries with each of the *countedTags* values mapping to thier counts.

Parameters

keyTag : `str`

The tag used as the key for the returned dictionary

*countedTags : `str, str, str, ...`

The tags used as the key for the returned dictionary's values

Returns

`dict[str:dict[str:int]]`

The dictionary of counts

discardID(*idVal*)
Checks if the collected items contains the give *idVal* and discards it if it is found, will not raise an exception if item is not found

Parameters

idVal : `str`

The discarded id string

dropBadEntries()
Removes all the bad entries from the collection

getID(*idVal*)
Looks up an item with *idVal* and returns it if it is found, returns `None` if it does not find the item

Parameters

idVal : `str`

The requested item's id string

Returns

`object`

The requested object or `None`

glimpse(**tags*, *compact=False*)
Creates a printable table with the most frequently occurring values of each of the requested *tags*, or if none are provided the top authors, journals and citations. The table will be as wide and as tall as the terminal (or 80x24 if there is no terminal) so `print(RC.glimpse())` should always create a nice looking table. Below is a table created from some of the testing files:

```
>>> print(RC.glimpse())
+RecordCollection glimpse made at: 2016-01-01␣
↪12:00:00++++++++++++++++++++++++++
|33 Records from␣
↪testFile++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|
|Columns are ranked by num. of occurrences and are independent of one␣
↪another++|
|-------Top Authors--------+------Top Journals-------+--------Top Cited-------
↪-|
|1                Girard, S|1 CANADIAN JOURNAL OF PH.|1 LEVY Y, 1975, OPT␣
↪COMM.|
|1                Gilles, H|1 JOURNAL OF THE OPTICAL.|2 GOOS F, 1947, ANN␣
↪PHYS.|
|2                IMBERT, C|2          APPLIED OPTICS|3 LOTSCH HKV, 1970,␣
↪OPTI.|
|2                Pillon, F|2   OPTICS COMMUNICATIONS|4 RENARD RH, 1964, J␣
↪OPT.|
|3          BEAUREGARD, OCD|2 NUOVO CIMENTO DELLA SO.|5 IMBERT C, 1972, PHYS␣
↪R.|
|3               Laroche, M|2 JOURNAL OF THE OPTICAL.|6 ARTMANN K, 1948, ANN␣
↪P.|
|3                 HUARD, S|2 JOURNAL OF THE OPTICAL.|6 COSTADEB.O, 1973,␣
↪PHYS.|
|4                  PURI, A|2 NOUVELLE REVUE D OPTIQ.|6 ROOSEN G, 1973, CR␣
↪ACA.|
|4               COSTADEB.O|3 PHYSICS REPORTS-REVIEW.|7 Imbert C., 1972,␣
↪Nouve.|
```

(continues on next page)

(continued from previous page)

```
|4           PATTANAYAK, DN|3 PHYSICAL REVIEW LETTERS|8 HOROWITZ BR, 1971, J␣
↪O.|
|4           Gazibegovic, A|3 USPEKHI FIZICHESKIKH N.|8 BRETENAKER F, 1992,␣
↪PH.|
|4                ROOSEN, G|3 APPLIED PHYSICS B-LASE.|8 SCHILLIN.H, 1965, ANN␣
↪.|
|4              BIRMAN, JL|3 AEU-INTERNATIONAL JOUR.|8 FEDOROV FI, 1955,␣
↪DOKL.|
|4               Kaiser, R|3 COMPTES RENDUS HEBDOMA.|8 MAZET A, 1971, CR␣
↪ACAD.|
|5                 LEVY, Y|3 CHINESE PHYSICS LETTERS|9 IMBERT C, 1972, CR␣
↪ACA.|
|5              BEAUREGA.OC|3       PHYSICAL REVIEW B|9 LOTSCH HKV, 1971,␣
↪OPTI.|
|5               PAVLOV, VI|3 LETTERE AL NUOVO CIMEN.|9 ASHBY N, 1973, PHYS␣
↪RE.|
|5                BREVIK, I|3 PROGRESS IN QUANTUM EL.|9 BOULWARE DG, 1973,␣
↪PHY.|
>>>
```

Parameters

tags : `str, str, ...`

Any number of tag strings to be made into columns in the output table

Returns

`str`

A string containing the table

networkMultiLevel(**modes*, *nodeCount=True*, *edgeWeight=True*, *stemmer=None*, *edgeAttribute=None*, *nodeAttribute=None*, *_networkTypeString='n-level network'*)

Creates a network of the objects found by any number of tags *modes*, with edges between all co-occurring values. IF you only want edges between co-occurring values from different tags use *networkMultiMode()*.

A **networkMultiLevel**() looks are each entry in the collection and extracts its values for the tag given by each of the *modes*, e.g. the `'authorsFull'` tag. Then if multiple are returned an edge is created between them. So in the case of the author tag `'authorsFull'` a co-authorship network is created. Then for each other tag the entries are also added and edges between the first tag's node and theirs are created.

The number of times each object occurs is count if *nodeCount* is `True` and the edges count the number of co-occurrences if *edgeWeight* is `True`. Both are`True` by default.

Note Do not use this for the construction of co-citation networks use Recordcollection.networkCoCitation() it is more accurate and has more options.

Parameters

mode : `str`

A two character WOS tag or one of the full names for a tag

nodeCount : `optional [bool]`

Default `True`, if `True` each node will have an attribute called "count" that contains an int giving the number of time the object occurred.

edgeWeight : `optional [bool]`

Default `True`, if `True` each edge will have an attribute called "weight" that contains an int giving the number of time the two objects co-occurrenced.

stemmer : `optional [func]`

Default `None`, If *stemmer* is a callable object, basically a function or possibly a class, it will be called for the ID of every node in the graph, all IDs are strings. For example:

The function `f = lambda x: x[0]` if given as the stemmer will cause all IDs to be the first character of their unstemmed IDs. e.g. the title `'Goos–Hanchen and Imbert–Fedorov shifts for leaky guided modes'` will create the node `'G'`.

Returns

`networkx Graph`

A networkx Graph with the objects of the tag *mode* as nodes and their co-occurrences as edges

networkMultiMode(**tags*, *recordType=True*, *nodeCount=True*, *edgeWeight=True*, *stemmer=None*, *edgeAttribute=None*)

Creates a network of the objects found by all tags in *tags*, each node is marked by which tag spawned it making the resultant graph n-partite.

A **networkMultiMode()** looks are each item in the collection and extracts its values for the tags given by *tags*. Then for all objects returned an edge is created between them, regardless of their type. Each node will have an attribute call `'type'` that gives the tag that created it or both if both created it, e.g. if `'LA'` were in *tags* node `'English'` would have the type attribute be `'LA'`.

For example if *tags* was set to `['CR', 'UT', 'LA']`, a three mode network would be created, composed of a co-citation network from the `'CR'` tag. Then each citation would also have edges to all the languages of Records that cited it and to the WOS number of the those Records.

The number of times each object occurs is count if *nodeCount* is `True` and the edges count the number of co-occurrences if *edgeWeight* is `True`. Both are`True` by default.

Parameters

tags : `str`, `str`, `str`, ... or `list [str]`

Any number of tags, or a list of tags

nodeCount : `optional [bool]`

Default `True`, if `True` each node will have an attribute called `'count'` that contains an int giving the number of time the object occurred.

edgeWeight : `optional [bool]`

Default `True`, if `True` each edge will have an attribute called `'weight'` that contains an int giving the number of time the two objects co-occurrenced.

stemmer : `optional [func]`

Default `None`, If *stemmer* is a callable object, basically a function or possibly a class, it will be called for the ID of every node in the graph, note that all IDs are strings.

For example: the function `f = lambda x:  x[0]` if given as the stemmer will cause all IDs to be the first character of their unstemmed IDs. e.g. the title `'Goos–Hanchen and Imbert–Fedorov shifts for leaky guided modes'` will create the node `'G'`.

Returns

`networkx Graph`

A networkx Graph with the objects of the tags *tags* as nodes and their co-occurrences as edges

networkOneMode(*mode*, *nodeCount=True*, *edgeWeight=True*, *stemmer=None*, *edgeAttribute=None*, *nodeAttribute=None*)

Creates a network of the objects found by one tag *mode*. This is the same as *networkMultiLevel()* with only one tag.

A **networkOneMode**() looks are each entry in the collection and extracts its values for the tag given by *mode*, e.g. the `'authorsFull'` tag. Then if multiple are returned an edge is created between them. So in the case of the author tag `'authorsFull'` a co-authorship network is created.

The number of times each object occurs is count if *nodeCount* is `True` and the edges count the number of co-occurrences if *edgeWeight* is `True`. Both are`True` by default.

Note Do not use this for the construction of co-citation networks use Recordcollection.networkCoCitation() it is more accurate and has more options.

Parameters

mode : `str`

A two character WOS tag or one of the full names for a tag

nodeCount : `optional [bool]`

Default `True`, if `True` each node will have an attribute called "count" that contains an int giving the number of time the object occurred.

edgeWeight : `optional [bool]`

Default `True`, if `True` each edge will have an attribute called "weight" that contains an int giving the number of time the two objects co-occurrenced.

stemmer : `optional [func]`

Default `None`, If *stemmer* is a callable object, basically a function or possibly a class, it will be called for the ID of every node in the graph, all IDs are strings. For example:

The function `f = lambda x:  x[0]` if given as the stemmer will cause all IDs to be the first character of their unstemmed IDs. e.g. the title `'Goos–Hanchen and Imbert–Fedorov shifts for leaky guided modes'` will create the node `'G'`.

Returns

`networkx Graph`

A networkx Graph with the objects of the tag *mode* as nodes and their co-occurrences as edges

networkTwoMode(*tag1*, *tag2*, *directed=False*, *recordType=True*, *nodeCount=True*, *edgeWeight=True*, *stemmerTag1=None*, *stemmerTag2=None*, *edgeAttribute=None*)

Creates a network of the objects found by two WOS tags *tag1* and *tag2*, each node marked by which tag spawned it making the resultant graph bipartite.

A **networkTwoMode()** looks at each Record in the `RecordCollection` and extracts its values for the tags given by *tag1* and *tag2*, e.g. the `'WC'` and `'LA'` tags. Then for each object returned by each tag and edge is created between it and every other object of the other tag. So the WOS defined subject tag `'WC'` and language tag `'LA'`, will give a two-mode network showing the connections between subjects and languages. Each node will have an attribute call `'type'` that gives the tag that created it or both if both created it, e.g. the node `'English'` would have the type attribute be `'LA'`.

The number of times each object occurs is count if *nodeCount* is `True` and the edges count the number of co-occurrences if *edgeWeight* is `True`. Both are`True` by default.

The *directed* parameter if `True` will cause the network to be directed with the first tag as the source and the second as the destination.

Parameters

tag1 : `str`

A two character WOS tag or one of the full names for a tag, the source of edges on the graph

tag1 : `str`

A two character WOS tag or one of the full names for a tag, the target of edges on the graph

directed : `optional [bool]`

Default `False`, if `True` the returned network is directed

nodeCount : `optional [bool]`

Default `True`, if `True` each node will have an attribute called "count" that contains an int giving the number of time the object occurred.

edgeWeight : `optional [bool]`

Default `True`, if `True` each edge will have an attribute called "weight" that contains an int giving the number of time the two objects co-occurrenced.

stemmerTag1 : `optional [func]`

Default `None`, If *stemmerTag1* is a callable object, basically a function or possibly a class, it will be called for the ID of every node given by *tag1* in the graph, all IDs are strings.

For example: the function `f = lambda x: x[0]` if given as the stemmer will cause all IDs to be the first character of their unstemmed IDs. e.g. the title `'Goos–Hanchen and Imbert–Fedorov shifts for leaky guided modes'` will create the node `'G'`.

stemmerTag2 : `optional [func]`

Default `None`, see *stemmerTag1* as it is the same but for *tag2*

Returns

`networkx Graph or networkx DiGraph`

A networkx Graph with the objects of the tags *tag1* and *tag2* as nodes and their co-occurrences as edges.

rankedSeries(*tag*, *outputFile=None*, *giveCounts=True*, *giveRanks=False*, *greatestFirst=True*, *pandasMode=True*, *limitTo=None*)

Creates an pandas dict of the ordered list of all the values of *tag*, with and ranked by their number of occurrences. A list can also be returned with the the counts or ranks added or it can be written to a file.

Parameters

tag : `str`

The tag to be ranked

outputFile : `optional str`

A file path to write a csv with 2 columns, one the tag values the other their counts

giveCounts : `optional bool`

Default `True`, if `True` the retuned list will be composed of tuples the first values being the tag value and the second their counts. This supersedes *giveRanks*.

giveRanks : `optional bool`

Default `False`, if `True` and *giveCounts* is `False`, the retuned list will be composed of tuples the first values being the tag value and the second their ranks. This is superseded by *giveCounts*.

greatestFirst : `optional bool`

Default `True`, if `True` the returned list will be ordered with the highest ranked value first, otherwise the lowest ranked will be first.

pandasMode : `optional bool`

Default `True`, if `True` a `dict` ready for pandas will be returned, otherwise a list

limitTo : `optional list[values]`

Default `None`, if a list is provided only those values in the list will be counted or returned

Returns

`dict[str:list[value]] or list[str]`

A `dict` or `list` will be returned depending on if *pandasMode* is `True`

removeID(*idVal*)

Checks if the collected items contains the give *idVal* and removes it if it is found, will raise a `KeyError` if item is not found

Parameters

idVal : `str`

The removed id string

tags()

Creates a list of all the tags of the contained items

Returns

`list [str]`

A list of all the tags

timeSeries(*tag=None*, *outputFile=None*, *giveYears=True*, *greatestFirst=True*, *limitTo=False*, *pandasMode=True*)

Creates an pandas dict of the ordered list of all the values of *tag*, with and ranked by the year the occurred in, multiple year occurrences will create multiple entries. A list can also be returned with the the counts or years added or it can be written to a file.

If no *tag* is given the `Records` in the collection will be used

Parameters

tag : `optional str`

Default `None`, if provided the tag will be ordered

outputFile : `optional str`

A file path to write a csv with 2 columns, one the tag values the other their years

giveYears : `optional bool`

Default `True`, if `True` the retuned list will be composed of tuples the first values being the tag value and the second their years.

greatestFirst : `optional bool`

Default `True`, if `True` the returned list will be ordered with the highest years first, otherwise the lowest years will be first.

pandasMode : `optional bool`

Default `True`, if `True` a `dict` ready for pandas will be returned, otherwise a list

limitTo : `optional list[values]`

Default `None`, if a list is provided only those values in the list will be counted or returned

Returns

`dict[str:list[value]] or list[str]`

A `dict` or `list` will be returned depending on if *pandasMode* is `True`

ExtendedRecord(Record)

class metaknowledge.**ExtendedRecord**(*fieldDict*, *idValue*, *bad*, *error*, *sFile=''*, *sLine=0*)

A subclass of `Record` that adds processing to the dictionary. It also cannot be use directly and must be subclassed.

The `ExtendedRecord` class is a extension of `Record` that is intended for use with the records on scientific papers provided by different organizations such as WOS or Pubmed. The 5 abstract (virtual) methods must be defined for each subclass and define how the data in the different fields is processed and how the record can be rewritten to a file.

Processing fields

When an `ExtendedRecord` is created a dictionary, *fieldDict*, must be provided this contains the raw data from the file reader, usually as lists of strings. `tagProcessingFunc` is a `staticmethod` function that takes in a tag string an returns another function to process it.

Each tag may also be given a second name, as usually what the they are called in the raw data are not very easy to understand (e.g. `'SO'` is the journal name for WOs records). The mapping from the raw tag (`'SO'`) to the human friendly string (`'journal'`) is done with the `getAltName` `staticmethod`. `getAltName` takes in a tag string and returns either `None` or the other name for that string. Note, `getAltName` must go both directions `WOSRecord.getAltName(WOSRecord.getAltName('SO')) == 'SO'`.

The last method for processing entries is `specialFuncs` The following are the special keys for `ExtendedRecords`. These must be the alternate names of tags or strings accepted by the `specialFuncs` method.

- `'authorsFull'`
- `'keywords'`
- `'grants'`
- `'j9'`
- `'authorsShort'`
- `'volume'`
- `'selfCitation'`
- `'citations'`
- `'address'`
- `'abstract'`
- `'title'`
- `'month'`
- `'year'`
- `'journal'`
- `'beginningPage'`
- `'DOI'`

`specialFuncs` when given one of these must raise a `KeyError` or return an object of the same type as that returned by the `MedlineRecord` or `WOSRecord`. e.g. `'title'` would return a string giving the title of the record.

For an example of how this works lets first look at the `'SO'` tag on a `WOSRecord` accessed with the alternate name `'journal'`.

```
t = R['journal']
```

First the private dictionary `_computedFields` is checked for the key `'title'`, which will fail if this is the first time `'journal'` or `'SO'` has been requested, after this the results will be added to the dictionary to speed up future requests.

Then the *fieldDict* will be checked for the key and when that fails the key will go through `getAltName` and be checked again. If the record had a journal entry this will succeed and the raw data will be given to the `tagProcessingFunc` using the same key as *fieldDict*, in this case `SO`.

The results will then be written to `_computedFields` and returned.

If the requested key was instead `'grants'` (`g = R['grants']`)the both lookups to *fieldDict* would have failed and the string `'grants'` would have been given to `specialFuncs` which would return a list of all the grants in the `WOSRecord` (this is always `[]` as WOS does not provided grant information).

What if the key were not present anywhere? Then the `specialFuncs` should raise a `KeyError` which will be caught then re-raised like a dictionary would with an invalid key look up.

File Handling fields

The two other required methods `encoding` and `writeRecord` define how the records can be rewritten to a file. `encoding` is should return a string giving the encoding python would use, e.g. `'utf-8'` or `'latin-1'`. This is the same encoding that the files written by `writeRecord` should have, `writeRecord` when called should write the original record to the provided open file, *infile*. The opening, closing, header and footer of the file will be handled by `RecordCollection`'s `writeFile` function which should me modified accordingly. If the order of the fields in a record is important you can use a collections.OrderedDict for *fieldDict*.

__Init__

The `__init__` of `ExtendedRecord` takes the same arguments as Record

__contains__(*item*)
Checks if the tag *item* is in the Record

__getitem__(*key*)
Processes the tag requested with *key* and memoize it.

Allows long names, but will still raise a KeyError if the tag is missing, regardless of name used.

__init__(*fieldDict*, *idValue*, *bad*, *error*, *sFile=''*, *sLine=0*)
Base constructor for Records

fieldDict : is the unpared entry dict with tags as keys and their lines as a list of strings

idValue : is the unique ID of the Record, e.g. the WOS number

titleKey : is the tag giving the title of the Record, e.g. the WOS tag is `'TI'`

bad : is the bool to flag the Record as having encountered an errror

error : is the error that bad indicates

sFile : is the name of the source file

sLine : is the line number of the start of the Record entry

altNames : is a dict that maps the names of tags to an alternative name, i.e. the long names dict. It **must** be bidirectional: map long to short and short to long

proccessingFuncs : is a dict of functions to proccess the tags. It has the short names as keys and their proccessing fucntions as values. Missing tags will result in the unparsed value to be returned.

The Records inheting from this must implement, calling the implementations in Record with super() will not cause errors:

- writeRecord
- tagProcessingFunc
- encoding

- titleTag
- getAltName

authGenders(*countsOnly=False*, *fractionsMode=False*, *_countsTuple=False*)
Creates a dict mapping `'Male'`, `'Female'` and `'Unknown'` to lists of the names of all the authors.

bibString(*maxLength=1000*, *WOSMode=False*, *restrictedOutput=False*, *niceID=True*)
Makes a string giving the Record as a bibTex entry. If the Record is of a journal article (`PT J`) the bibtext type is set to `'article'`, otherwise it is set to `'misc'`. The ID of the entry is the WOS number and all the Record's fields are given as entries with their long names.

Note This is not meant to be used directly with LaTeX none of the special characters have been escaped and there are a large number of unnecessary fields provided. *niceID* and *maxLength* have been provided to make conversions easier.

Note Record entries that are lists have their values seperated with the string `' and '`

createCitation(*multiCite=False*)
Creates a citation string, using the same format as other WOS citations, for the Record by reading the relevant special tags (`'year'`, `'J9'`, `'volume'`, `'beginningPage'`, `'DOI'`) and using it to create a Citation object.

encoding()
An `abstractmethod`, gives the encoding string of the record.

get(*tag*, *default=None*, *raw=False*)
Allows access to the raw values or is an Exception safe wrapper to `__getitem__`.

static getAltName(*tag*)
An `abstractmethod`, gives the alternate name of *tag* or `None`

getCitations(*field=None*, *values=None*, *pandasFriendly=True*)
Creates a pandas ready dict with each row a different citation and columns containing the original string, year, journal and author's name.

There are also options to filter the output citations with *field* and *values*

items(*raw=False*)
Like `items` for dicts but with a `raw` option

specialFuncs(*key*)
An `abstractmethod`, process the special tag, *key* using the whole `Record`

subDict(*tags*, *raw=False*)
Creates a dict of values of *tags* from the Record. The tags are the keys and the values are the values. If the tag is missing the value will be `None`.

static tagProcessingFunc(*tag*)
An `abstractmethod`, gives the function for processing *tag*

values(*raw=False*)
Like `values` for dicts but with a `raw` option

writeRecord(*infile*)
An `abstractmethod`, writes the record in its original form to *infile*

FallbackGrant(Grant)

class metaknowledge.grants.**FallbackGrant**(*original*, *grantdDict*, *sFile=''*, *sLine=0*)
A subclass of Grant, it has the same attributes and is returned from the fall back constructor for grants.

__init__ (*original*, *grantdDict*, *sFile=''*, *sLine=0*)
Initialize self. See help(type(self)) for accurate signature.

Grant(Record, MutableMapping)

class metaknowledge.grants.**Grant** (*original*, *grantdDict*, *idValue*, *bad*, *error*, *sFile=''*, *sLine=0*)

__init__ (*original*, *grantdDict*, *idValue*, *bad*, *error*, *sFile=''*, *sLine=0*)
Initialize self. See help(type(self)) for accurate signature.

getInstitutions (*tags=None*, *seperator=';'*, *_getTag=False*)
Returns a list of the names of institutions. This is done by looking (in order) for any of fields in *tags* and splitting the strings on *seperator* (in case of multiple institutions). If no strings are found an empty list will be returned.

Note for some Grants `getInstitutions` has been overwritten and will ignore the arguments and simply provide the investigators.

Parameters

tags : `optional list[str]`

A list of the tags to look for institutions in

seperator : `optional str`

The string that separators each institutions name within the column

Returns

`list [str]`

A list of all the found institution's names

getInvestigators (*tags=None*, *seperator=';'*, *_getTag=False*)
Returns a list of the names of investigators. This is done by looking (in order) for any of fields in *tags* and splitting the strings on *seperator*. If no strings are found an empty list will be returned.

Note for some Grants `getInvestigators` has been overwritten and will ignore the arguments and simply provide the investigators.

Parameters

tags : `optional list[str]`

A list of the tags to look for investigators in

seperator : `optional str`

The string that separators each investigators name within the column

Returns

`list [str]`

A list of all the found investigator's names

update(*other*)
Adds all the tag-entry pairs from *other* to the `Grant`. If there is a conflict *other* takes precedence.

Parameters

other : `Grant`

Another `Grant` of the same type as *self*

GrantCollection(CollectionWithIDs)

class metaknowledge.**GrantCollection**(*inGrants=None*, *name=''*, *extension=''*, *cached=False*, *quietStart=False*)

__init__(*inGrants=None*, *name=''*, *extension=''*, *cached=False*, *quietStart=False*)
Basically a collections.abc.MutableSet wrapper for a set with a bunch of extra record keeping attached.

networkCoInvestigator(*targetTags=None*, *tagSeperator=';'*, *count=True*, *weighted=True*, *_institutionLevel=False*)
Creates a co-investigator from the collection

Most grants do not have a known investigator tag so it must be provided by the user in *targetTags* and the separator character if it is not a semicolon should also be given.

Parameters

targetTags : `optional list[str]`

A list of all the Grant tags to check for investigators

tagSeperator : `optional str`

The character that separates the individual investigator's names

count : `optional bool`

Default `True`, if `True` the number of time a name occurs will be given

weighted : `optional bool`

Default `True`, if `True` the edge weights will be calculated and added to the edges

Returns

`networkx Graph`

The graph of co-investigator

networkCoInvestigatorInstitution(*targetTags=None*, *tagSeperator=';'*, *count=True*, *weighted=True*)
This works the same as *networkCoInvestigator()* see it for details.

MedlineGrant(Grant)

class metaknowledge.**MedlineGrant**(*grantString*)

__init__(*grantString*)
Initialize self. See help(type(self)) for accurate signature.

MedlineRecord(ExtendedRecord)

class metaknowledge.medline.**MedlineRecord**(*inRecord*, *sFile=''*, *sLine=0*)
Class for full Medline(Pubmed) entries.

This class is an ExtendedRecord capable of generating its own id number. You should not create them directly, but instead use medlineParser() on a medline file.

__init__(*inRecord*, *sFile=''*, *sLine=0*)
Base constructor for Records

fieldDict : is the unpared entry dict with tags as keys and their lines as a list of strings

idValue : is the unique ID of the Record, e.g. the WOS number

titleKey : is the tag giving the title of the Record, e.g. the WOS tag is `'TI'`

bad : is the bool to flag the Record as having encountered an errror

error : is the error that bad indicates

sFile : is the name of the source file

sLine : is the line number of the start of the Record entry

altNames : is a dict that maps the names of tags to an alternative name, i.e. the long names dict. It **must** be bidirectional: map long to short and short to long

proccessingFuncs : is a dict of functions to proccess the tags. It has the short names as keys and their proccessing fucntions as values. Missing tags will result in the unparsed value to be returned.

The Records inheting from this must implement, calling the implementations in Record with super() will not cause errors:

- writeRecord
- tagProcessingFunc
- encoding
- titleTag
- getAltName

encoding()
An abstractmethod, gives the encoding string of the record.

Returns

str

The encoding

static getAltName(*tag*)
An abstractmethod, gives the alternate name of *tag* or None

Parameters

tag : `str`

The requested tag

Returns

`str`

The alternate name of *tag* or `None`

specialFuncs(*key*)
An `abstractmethod`, process the special tag, *key* using the whole `Record`

Parameters

key : `str`

One of the special tags: `'authorsFull'`, `'keywords'`, `'grants'`, `'j9'`, `'authorsShort'`, `'volume'`, `'selfCitation'`, `'citations'`, `'address'`, `'abstract'`, `'title'`, `'month'`, `'year'`, `'journal'`, `'beginningPage'` and `'DOI'`

Returns

The processed value of *key*

static tagProcessingFunc(*tag*)
An `abstractmethod`, gives the function for processing *tag*

Parameters

tag : `optional [str]`

The tag in need of processing

Returns

`function`

The function to process the raw tag

writeRecord(*f*)
This is nearly identical to the original the FAU tag is the only tag not writen in the same place, doing so would require changing the parser and lots of extra logic.

NSERCGrant(Grant)

class metaknowledge.grants.**NSERCGrant**(*original*, *grantdDict*, *sFile*, *sLine*)

__init__(*original*, *grantdDict*, *sFile*, *sLine*)
Initialize self. See help(type(self)) for accurate signature.

getInstitutions(*tags=None*, *seperator=';'*, *_getTag=False*)
Returns a list with the names of the institution. The optional arguments are ignored

Returns

`list [str]`

A list with 1 entry the name of the institution

getInvestigators(*tags=None*, *seperator=';'*, *_getTag=False*)
Returns a list of the names of investigators. The optional arguments are ignored.

Returns

`list [str]`

A list of all the found investigator's names

update(*other*)
Adds all the tag-entry pairs from *other* to the `Grant`. If there is a conflict *other* takes precedence.

Parameters

other : `Grant`

Another `Grant` of the same type as *self*

NSFGrant(Grant)

class metaknowledge.grants.**NSFGrant**(*grantdDict*, *sFile*)

__init__(*grantdDict*, *sFile*)
Initialize self. See help(type(self)) for accurate signature.

getInstitutions(*tags=None*, *seperator=';'*, *_getTag=False*)
Returns a list with the names of the institution. The optional arguments are ignored

Returns

`list [str]`

A list with 1 entry the name of the institution

getInvestigators(*tags=None*, *seperator=';'*, *_getTag=False*)
Returns a list of the names of investigators. The optional arguments are ignored.

Returns

`list [str]`

A list of all the found investigator's names

ProQuestRecord(ExtendedRecord)

class metaknowledge.proquest.**ProQuestRecord**(*inRecord*, *recNum=None*, *sFile=''*, *sLine=0*)

Class for full ProQuest entries.

This class is an ExtendedRecord capable of generating its own id number. You should not create them directly, but instead use proQuestParser() on a ProQuest file.

__init__(*inRecord*, *recNum=None*, *sFile=''*, *sLine=0*)

Base constructor for Records

fieldDict : is the unpared entry dict with tags as keys and their lines as a list of strings

idValue : is the unique ID of the Record, e.g. the WOS number

titleKey : is the tag giving the title of the Record, e.g. the WOS tag is `'TI'`

bad : is the bool to flag the Record as having encountered an errror

error : is the error that bad indicates

sFile : is the name of the source file

sLine : is the line number of the start of the Record entry

altNames : is a dict that maps the names of tags to an alternative name, i.e. the long names dict. It **must** be bidirectional: map long to short and short to long

proccessingFuncs : is a dict of functions to proccess the tags. It has the short names as keys and their proccessing fucntions as values. Missing tags will result in the unparsed value to be returned.

The Records inheting from this must implement, calling the implementations in Record with super() will not cause errors:

- writeRecord
- tagProcessingFunc
- encoding
- titleTag
- getAltName

encoding()

An `abstractmethod`, gives the encoding string of the record.

Returns

`str`

The encoding

static getAltName(*tag*)

An `abstractmethod`, gives the alternate name of *tag* or `None`

Parameters

tag : `str`

The requested tag

Returns

`str`

The alternate name of *tag* or `None`

specialFuncs(*key*)
An `abstractmethod`, process the special tag, *key* using the whole `Record`

Parameters

key : `str`

One of the special tags: `'authorsFull'`, `'keywords'`, `'grants'`, `'j9'`, `'authorsShort'`, `'volume'`, `'selfCitation'`, `'citations'`, `'address'`, `'abstract'`, `'title'`, `'month'`, `'year'`, `'journal'`, `'beginningPage'` and `'DOI'`

Returns

The processed value of *key*

static tagProcessingFunc(*tag*)
An `abstractmethod`, gives the function for processing *tag*

Parameters

tag : `optional [str]`

The tag in need of processing

Returns

`function`

The function to process the raw tag

writeRecord(*infile*)
An `abstractmethod`, writes the record in its original form to *infile*

Parameters

infile : `writable file`

The file to be written to

Record(Mapping, Hashable)

class metaknowledge.**Record**(*fieldDict*, *idValue*, *bad*, *error*, *sFile=''*, *sLine=0*)
A dictionary with error handling and an id string.

`Record` is the base class of the all objects in *metaknowledge* that contain information as key-value pairs, these are the grants and the records from different sources.

The error handling of the `Record` is done with the `bad` attribute. If there is some issue with the data *bad* should be `True` and *error* given an `Exception` that was caused by or explains the error.

Customizations

`Record` is a subclass of `abc.collections.Mapping` which means it has almost all the methods a dictionary does, the missing ones are those that modify entries. So to access the value of the key `'title'` from a `Record` `R`, you would use either the square brace notation `t = R['title']` or the `get()` function `t = R.get('title')` just like a dictionary. The other methods like `keys()` or `copy()` also work.

In addition to being a mapping `Records` are also hashable with their hashes being based on a unique id string they are given on creation, usually some kind of accession number the source gives them. The two optional arguments *sFile* and *sLine*, which should be given the name of the file the records came from and the line it started on respectively, are used to make the errors more useful.

__Init__

fieldDict is the dictionary the `Record` will use and *idValue* is the unique identifier of the `Record`.

Parameters

fieldDict : `dict[str:]`

A dictionary that maps from strings to values

idValue : `str`

A unique identifier string for the `Record`

bad : `bool`

`True` if there are issues with the `Record`, otherwise `False`

error : `Exception`

The `Exception` that caused whatever error made the record be marked as bad or `None`

sFile : `str`

A string that gives the source file of the original records

sLine : `int`

The first line the original record is found on in the source file

__bytes__()
Returns the binary form of the original

__contains__(*item*)
Checks if the tag *item* is in the Record

__eq__(*other*)
Compares `Records` using their hashes if their hashes are the same then `True` is returned.

__getitem__(*key*)
This is redfined as something interesting for ExtendedRecord

__hash__()
Gives a hash of the id or if `bad` returns a hash of the fields combined with the error messages, either of these could be blank

`bad` Records are more likely to cause hash collisions due to their lack of entropy when created.

__init__(*fieldDict*, *idValue*, *bad*, *error*, *sFile=''*, *sLine=0*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()
Iterates over the tags in the Record

__len__()
Returns the number of tags

__repr__()
Makes a string with the id of the file and its type

__str__()
Makes a string with the title of the file as given by self.title, if there is not one it returns "Untitled record"

__weakref__
list of weak references to the object (if defined)

copy()
Correctly copies the `Record`

RecordCollection(CollectionWithIDs)

class metaknowledge.**RecordCollection**(*inCollection=None*, *name=''*, *extension=''*, *cached=False*, *quietStart=False*)

A container for a large number of indivual records.

`RecordCollection` provides ways of creating Records from an isi file, string, list of records or directory containing isi files.

When being created if there are issues the Record collection will be declared bad, `bad` wil be set to `False`, it will then mostly return `None` or False. The attribute `error` contains the exception that occurred.

They also possess an attribute `name` also accessed with `__repr__()`, this is used to auto generate the names of files and can be set at creation, note though that any operations that modify the RecordCollection's contents will update the name to include what occurred.

Customizations

The Records are containing within a set and as such many of the set operations are defined, pop, union, in ... also records are hashed with their WOS string so no duplication can occur. The comparison operators <, <=, >, >= are based strictly on the number of Records within the collection, while equality looks for an exact match on the Records

__Init__

inCollection is the object containing the information about the Records to be constructed it can be an isi file, string, list of records or directory containing isi files

Parameters

inCollection : `optional [str] or None`

the name of the source of WOS records. It can be skipped to produce an empty collection.

If a file is provided. First it is checked to see if it is a WOS file (the header is checked). Then records are read from it one by one until the 'EF' string is found indicating the end of the file.

If a directory is provided. First each file in the directory is checked for the correct header and all those that do are then read like indivual files. The records are then collected into a single set in the RecordCollection.

name : `optional [str]`

The name of the RecordCollection, defaults to empty string. If left empty the name of the Record collection is set to the name of the file or directory used to create the collection. If provided the name id set to *name*

extension : `optional [str]`

The extension to search for when reading a directory for files. *extension* is the suffix searched for when a directory is read for files, by default it is empty so all files are read.

cached : `optional [bool]`

Default `False`, if `True` and the *inCollection* is a directory (a string giving the path to a directory) then the initialized `RecordCollection` will be saved in the directory as a Python pickle with the suffix `'.mkDirCache'`. Then if the `RecordCollection` is initialized a second time it will be recovered from the file, which is much faster than reprising every file in the directory.

metaknowledge saves the names of the parsed files as well as their last modification times and will check these when recreating the `RecordCollection`, so modifying existing files or adding new ones will result in the entire directory being reanalyzed and a new cache file being created. The extension given to `__init__()` is taken into account as well and each suffix is given its own cache.

Note The pickle allows for arbitrary python code execution so only use caches that you trust.

__init__(*inCollection=None*, *name=''*, *extension=''*, *cached=False*, *quietStart=False*)
Basically a collections.abc.MutableSet wrapper for a set with a bunch of extra record keeping attached.

citeFilter(*keyString=''*, *field='all'*, *reverse=False*, *caseSensitive=False*)
Filters `Records` by some string, *keyString*, in their citations and returns all `Records` with at least one citation possessing *keyString* in the field given by *field*.

dropNonJournals(*ptVal='J'*, *dropBad=True*, *invert=False*)
Drops the non journal type `Records` from the collection, this is done by checking *ptVal* against the PT tag

findProbableCopyright()
Finds the (likely) copyright string from all abstracts in the `RecordCollection`

forBurst(*tag*, *outputFile=None*, *dropList=None*, *lower=True*, *removeNumbers=True*, *removeNonWords=True*, *removeWhitespace=True*, *stemmer=None*)
Creates a pandas friendly dictionary with 2 columns one `'year'` and the other `'word'`. Each row is a

word that occurred in the field given by *tag* in a `Record` and the year of the record. Unfortunately getting the month or day with any type of accuracy has proved to be impossible so year is the only option.

forNLP(*outputFile=None*, *extraColumns=None*, *dropList=None*, *lower=True*, *removeNumbers=True*, *removeNonWords=True*, *removeWhitespace=True*, *removeCopyright=False*, *stemmer=None*)
Creates a pandas friendly dictionary with each row a `Record` in the `RecordCollection` and the columns fields natural language processing uses (id, title, publication year, keywords and the abstract). The abstract is by default is processed to remove non-word, non-space characters and the case is lowered.

genderStats(*asFractions=False*)
Creates a dict (`{'Male' : maleCount, 'Female' : femaleCount, 'Unknown' : unknownCount}`) with the numbers of male, female and unknown names in the collection.

getCitations(*field=None*, *values=None*, *pandasFriendly=True*, *counts=True*)
Creates a pandas ready dict with each row a different citation the contained Records and columns containing the original string, year, journal, author's name and the number of times it occured.

There are also options to filter the output citations with *field* and *values*

localCiteStats(*pandasFriendly=False*, *keyType='citation'*)
Returns a dict with all the citations in the CR field as keys and the number of times they occur as the values

localCitesOf(*rec*)
Takes in a Record, WOS string, citation string or Citation and returns a RecordCollection of all records that cite it.

makeDict(*onlyTheseTags=None*, *longNames=False*, *raw=False*, *numAuthors=True*, *genderCounts=True*)
Returns a dict with each key a tag and the values being lists of the values for each of the Records in the collection, `None` is given when there is no value and they are in the same order across each tag.

When used with pandas: `pandas.DataFrame(RC.makeDict())` returns a data frame with each column a tag and each row a Record.

networkBibCoupling(*weighted=True*, *fullInfo=False*, *addCR=False*)
Creates a bibliographic coupling network based on citations for the RecordCollection.

networkCitation(*dropAnon=False*, *nodeType='full'*, *nodeInfo=True*, *fullInfo=False*, *weighted=True*, *dropNonJournals=False*, *count=True*, *directed=True*, *keyWords=None*, *detailedCore=True*, *detailedCoreAttributes=False*, *coreOnly=False*, *expandedCore=False*, *recordToCite=True*, *addCR=False*, *_quiet=False*)
Creates a citation network for the RecordCollection.

networkCoAuthor(*detailedInfo=False*, *weighted=True*, *dropNonJournals=False*, *count=True*, *useShortNames=False*, *citeProfile=False*)
Creates a coauthorship network for the RecordCollection.

networkCoCitation(*dropAnon=True*, *nodeType='full'*, *nodeInfo=True*, *fullInfo=False*, *weighted=True*, *dropNonJournals=False*, *count=True*, *keyWords=None*, *detailedCore=True*, *detailedCoreAttributes=False*, *coreOnly=False*, *expandedCore=False*, *addCR=False*)
Creates a co-citation network for the RecordCollection.

rpys(*minYear=None*, *maxYear=None*, *dropYears=None*, *rankEmptyYears=False*)
This implements *Referenced Publication Years Spectroscopy* a techinique for finding import years in citation data. The authors of the original papers have a website with more information, found here.

This function computes the spectra of the `RecordCollection` and returns a dictionary mapping strings to lists of `ints`. Each list is ordered and the values of each with the same index form a row and each list a column. The strings are the names of the columns. This is intended to be read directly by pandas `DataFrames`.

The columns returned are:

1. `'year'`, the years of the counted citations, missing years are inserted with a count of 0, unless they are outside the bounds of the highest year or the lowest year and the default value is used. e.g. if the highest year is 2016, 2017 will not be inserted unless *maxYear* has been set to 2017 or higher
2. `'count'`, the number of times the year was cited
3. `'abs-deviation'`, deviation from the 5-year median. Calculated by taking the absolute deviation of the count from the median of it and the next 2 years and the preceding 2 years
4. `'rank'`, the rank of the year, the highest ranked year being the one with the highest deviation, the second highest being the second highest deviation and so on. All years with 0 count are given the rank 0 by default

writeBib(*fname=None*, *maxStringLength=1000*, *wosMode=False*, *reducedOutput=False*, *niceIDs=True*)
Writes a bibTex entry to *fname* for each `Record` in the collection.

If the Record is of a journal article (PT J) the bibtext type is set to `'article'`, otherwise it is set to `'misc'`. The ID of the entry is the WOS number and all the Record's fields are given as entries with their long names.

Note This is not meant to be used directly with LaTeX none of the special characters have been escaped and there are a large number of unnecessary fields provided. *niceID* and *maxLength* have been provided to make conversions easier only.

Note Record entries that are lists have their values separated with the string `' and '`, as this is the way bibTex understands

writeCSV(*fname=None*, *splitByTag=None*, *onlyTheseTags=None*, *numAuthors=True*, *genderCounts=True*, *longNames=False*, *firstTags=None*, *csvDelimiter=','*, *csvQuote='"'*, *listDelimiter='|'*)
Writes all the `Records` from the collection into a csv file with each row a record and each column a tag.

writeFile(*fname=None*)
Writes the `RecordCollection` to a file, the written file's format is identical to those download from WOS. The order of `Records` written is random.

yearSplit(*startYear*, *endYear*, *dropMissingYears=True*)
Creates a RecordCollection of Records from the years between *startYear* and *endYear* inclusive.

ScopusRecord(ExtendedRecord)

class metaknowledge.scopus.**ScopusRecord**(*inRecord*, *sFile=''*, *sLine=0*, *header=None*)
Class for full Scopus entries.

This class is an ExtendedRecord capable of generating its own id number. You should not create them directly, but instead use scopusParser() on a scopus **CSV** file.

__init__(*inRecord*, *sFile=''*, *sLine=0*, *header=None*)
Base constructor for Records

fieldDict : is the unpared entry dict with tags as keys and their lines as a list of strings

idValue : is the unique ID of the Record, e.g. the WOS number

titleKey : is the tag giving the title of the Record, e.g. the WOS tag is `'TI'`

bad : is the bool to flag the Record as having encountered an errror

error : is the error that bad indicates

sFile : is the name of the source file

sLine : is the line number of the start of the Record entry

altNames : is a dict that maps the names of tags to an alternative name, i.e. the long names dict. It **must** be bidirectional: map long to short and short to long

proccessingFuncs : is a dict of functions to proccess the tags. It has the short names as keys and their proccessing fucntions as values. Missing tags will result in the unparsed value to be returned.

The Records inheting from this must implement, calling the implementations in Record with super() will not cause errors:

- writeRecord
- tagProcessingFunc
- encoding
- titleTag
- getAltName

createCitation(*multiCite=False*)

Overwriting the general citation creator to deal with scopus weirdness.

Creates a citation string, using the same format as other WOS citations, for the Record by reading the relevant special tags (`'year'`, `'J9'`, `'volume'`, `'beginningPage'`, `'DOI'`) and using it to create a Citation object.

Parameters

multiCite : `optional [bool]`

Default `False`, if `True` a tuple of Citations is returned with each having a different one of the records authors as the author

Returns

`Citation`

A Citation object containing a citation for the Record.

encoding()

An `abstractmethod`, gives the encoding string of the record.

Returns

`str`

The encoding

static getAltName(*tag*)

An `abstractmethod`, gives the alternate name of *tag* or `None`

Parameters

tag : `str`

The requested tag

Returns

`str`

The alternate name of *tag* or `None`

specialFuncs(*key*)
An `abstractmethod`, process the special tag, *key* using the whole `Record`

Parameters

key : `str`

One of the special tags: `'authorsFull'`, `'keywords'`, `'grants'`, `'j9'`, `'authorsShort'`, `'volume'`, `'selfCitation'`, `'citations'`, `'address'`, `'abstract'`, `'title'`, `'month'`, `'year'`, `'journal'`, `'beginningPage'` and `'DOI'`

Returns

The processed value of *key*

static tagProcessingFunc(*tag*)
An `abstractmethod`, gives the function for processing *tag*

Parameters

tag : `optional [str]`

The tag in need of processing

Returns

`function`

The function to process the raw tag

writeRecord(*f*)
An `abstractmethod`, writes the record in its original form to *infile*

Parameters

infile : `writable file`

The file to be written to

WOSRecord(ExtendedRecord)

class metaknowledge.WOS.**WOSRecord**(*inRecord*, *sFile=''*, *sLine=0*)
Class for full WOS records

It is meant to be immutable; many of the methods and attributes are evaluated when first called, not when the object is created, and the results are stored privately.

The record's meta-data is stored in an ordered dictionary labeled by WOS tags. To access the raw data stored in the original record the tags() method can be used. To access data that has been processed and cleaned the attributes named after the tags are used.

Customizations

The `Record`'s hashing and equality testing are based on the WOS number (the tag is 'UT', and also called the accession number). They are strings starting with `'WOS:'` and followed by 15 or so numbers and letters, although both the length and character set are known to vary. The numbers are unique to each record so are used for comparisons. If a record is `bad` all equality checks return `False`.

When converted to a string the records title is used so for a record `R`, `R.TI == R.title == str(R)` and its representation uses the WOS number instead of memory location.

Attributes

When a record is created if the parsing of the WOS file failed it is marked as `bad`. The `bad` attribute is set to True and the `error` attribute is created to contain the exception object.

Generally, to get the information from a Record its attributes should be used. For a Record `R`, calling `R.CR` causes citations() from the the tagProcessing module to be called on the contents of the raw 'CR' field. Then the result is saved and returned. In this case, a list of Citation objects is returned. You can also call `R.citations` to get the same effect, as each known field tag has a longer name (currently there are 61 field tags). These names are meant to make accessing tags more readable and mapping from tag to name can be found in the tagToFull dict. If a tag is known (in tagToFull) but not in the raw data `None` is returned instead. Most tags when cleaned return a string or list of strings, the exact results can be found in the help for the particular function.

The attribute `authors` is also defined as a convenience and returns the same as 'AF' or if that is not found 'AU'.

__Init__

Records are generally created as collections in Recordcollections, and not as individual objects. If you wish to create one on its own it is possible, the arguments are as follows.

Parameters

inRecord: `files stream, dict, str or itertools.chain`

If it is a file stream the file must be open at the location of the first tag in the record, usually 'PT', and the file will be read until 'ER' is found, which indicates the end of the record in the file.

If a dict is passed the dictionary is used as the database of fields and tags, so each key is considered a WOS tag and each value a list of the lines of the original associated with the tag. This is the same form of dict that recordParser returns.

For a string the input must be the raw textual data of a single record in the WOS style, like the file stream it must start at the first tag and end in `'ER'`.

itertools.chain is treated identically to a file stream and is used by RecordCollections.

sFile : `optional [str]`

Is the name of the file the raw data was in, by default it is blank. It is mostly used to make error messages more informative.

sLine : `optional [int]`

Is the line the record starts on in the raw data file. It is mostly used to make error messages more informative.

UT
Returns the UT tag (WOS number) of the record

__init__(*inRecord*, *sFile=''*, *sLine=0*)
See help on Record for details

encoding()
An `abstractmethod`, gives the encoding string of the record.

static getAltName(*tag*)
An `abstractmethod`, gives the alternate name of *tag* or `None`

specialFuncs(*key*)
An `abstractmethod`, process the special tag, *key* using the whole `Record`

static tagProcessingFunc(*tag*)
An `abstractmethod`, gives the function for processing *tag*

wosString
Returns the WOS number (UT tag) of the record

writeRecord(*infile*)
Writes to *infile* the original contents of the Record. This is intended for use by RecordCollections to write to file. What is written to *infile* is bit for bit identical to the original record file (if utf-8 is used). No newline is inserted above the write but the last character is a newline.

3.2.5 Functions

`metaknowledge.citation.`**filterNonJournals**(*citesLst*, *invert=False*)
Removes the `Citations` from *citesLst* that are not journals

Parameters

citesLst : `list [Citation]`

A list of citations to be filtered

invert : `optional [bool]`

Default `False`, if `True` non-journals will be kept instead of journals

Returns

`list [Citation]`

A filtered list of Citations from *citesLst*

`metaknowledge.constants.`**`isInteractive`**`()`

A basic check of if the program is running in interactive mode

`metaknowledge.diffusion.`**`diffusionAddCountsFromSource`**`(`*grph*, *source*, *target*, *nodeType='citations'*, *extraType=None*, *diffusionLabel='DiffusionCount'*, *extraKeys=None*, *countsDict=None*, *extraMapping=None*`)`

Does a diffusion using *diffusionCount()* and updates *grph* with it, using the nodes in the graph as keys in the diffusion, i.e. the source. The name of the attribute the counts are added to is given by *diffusionLabel*. If the graph is not composed of citations from the source and instead is another tag *nodeType* needs to be given the tag string.

Parameters

grph : `networkx Graph`

The graph to be updated

source : `RecordCollection`

The `RecordCollection` that created *grph*

target : `RecordCollection`

The `RecordCollection` that will be counted

nodeType : `optional [str]`

default `'citations'`, the tag that constants the values used to create *grph*

Returns

`dict[:int]`

The counts dictioanry used to add values to *grph*. *Note grph* is modified by the function and the return is done in case you need it.

`metaknowledge.diffusion.`**`diffusionCount`**`(`*source*, *target*, *sourceType='raw'*, *extraValue=None*, *pandasFriendly=False*, *compareCounts=False*, *numAuthors=True*, *useAllAuthors=True*, *_ProgBar=None*, *extraMapping=None*`)`

Takes in two RecordCollections and produces a `dict` counting the citations of *source* by the Records of *target*. By default the `dict` uses `Record` objects as keys but this can be changed with the *sourceType* keyword to any of the WOS tags.

Parameters

source : `RecordCollection`

A metaknowledge `RecordCollection` containing the `Records` being cited

target : `RecordCollection`

A metaknowledge `RecordCollection` containing the `Records` citing those in *source*

sourceType : `optional [str]`

default `'raw'`, if `'raw'` the returned `dict` will contain `Records` as keys. If it is a WOS tag the keys will be of that type.

pandasFriendly : `optional [bool]`

default `False`, makes the output be a dict with two keys one `"Record"` is the list of Records (or data type requested by *sourceType*) the other is their occurrence counts as `"Counts"`. The lists are the same length.

compareCounts : `optional [bool]`

default `False`, if `True` the diffusion analysis will be run twice, first with source and target setup like the default (global scope) then using only the source `RecordCollection` (local scope).

extraValue : `optional [str]`

default `None`, if a tag the returned dictionary will have `Records` mapped to maps, these maps will map the entries for the tag to counts. If *pandasFriendly* is also `True` the resultant dictionary will have an additional column called `'year'`. This column will contain the year the citations occurred, in addition the Records entries will be duplicated for each year they occur in.

For example if `'year'` was given then the count for a single `Record` could be `{1990 : 1, 2000 : 5}`

useAllAuthors : `optional [bool]`

default `True`, if `False` only the first author will be used to generate the `Citations` for the *source* `Records`

Returns

`dict[:int]`

A dictionary with the type given by *sourceType* as keys and integers as values.

If *compareCounts* is `True` the values are tuples with the first integer being the diffusion in the target and the second the diffusion in the source.

If *pandasFriendly* is `True` the returned dict has keys with the names of the WOS tags and lists with their values, i.e. a table with labeled columns. The counts are in the column named `"TargetCount"` and if *compareCounts* the local count is in a column called `"SourceCount"`.

`metaknowledge.diffusion.`**`diffusionGraph`**(*source*, *target*, *weighted=True*, *sourceType='raw'*, *targetType='raw'*, *labelEdgesBy=None*)

Takes in two RecordCollections and produces a graph of the citations of *source* by the Records in *target*. By default the nodes in the are `Record` objects but this can be changed with the *sourceType* and *targetType* keywords. The edges of the graph go from the target to the source.

Each node on the output graph has two boolean attributes, `"source"` and `"target"` indicating if they are targets or sources. Note, if the types of the sources and targets are different the attributes will not be checked for overlap of the other type. e.g. if the source type is `'TI'` (title) and the target type is `'UT'` (WOS number), and there is some overlap of the targets and sources. Then the Record corresponding to a source node will not be checked for being one of the titles of the targets, only its WOS number will be considered.

Parameters

source : `RecordCollection`

A metaknowledge `RecordCollection` containing the `Records` being cited

target : `RecordCollection`

A metaknowledge `RecordCollection` containing the `Records` citing those in *source*

weighted : `optional [bool]`

Default `True`, if `True` each edge will have an attribute `'weight'` giving the number of times the source has referenced the target.

sourceType : `optional [str]`

Default `'raw'`, if `'raw'` the returned graph will contain `Records` as source nodes.

If Records are not wanted then it can be set to a WOS tag, such as `'SO'` (for journals), to make the nodes into the type of object returned by that tag from Records.

targetType : `optional [str]`

Default `'raw'`, if `'raw'` the returned graph will contain `Records` as target nodes.

If Records are not wanted then it can be set to a WOS tag, such as `'SO'` (for journals), to make the nodes into the type of object returned by that tag from Records.

labelEdgesBy : `optional [str]`

Default `None`, if a WOS tag (or long name of WOS tag) then the edges of the output graph will have a attribute `'key'` that is the value of the referenced tag, of source `Record`, i.e. if `'PY'` is given then each edge will have a `'key'` value equal to the publication year of the source.

This option will cause the output graph to be an `MultiDiGraph` and is likely to result in parallel edges. If a `Record` has multiple values for at tag (e.g. `'AF'`) the each tag will create its own edge.

Returns

`networkx Directed Graph or networkx multi Directed Graph`

A directed graph of the diffusion network, *labelEdgesBy* is used the graph will allow parallel edges.

`metaknowledge.diffusion.`**`makeNodeID`**(*Rec*, *ndType*, *extras=None*)

Helper to make a node ID, extras is currently not used

`metaknowledge.graphHelpers.`**`dropEdges`**(*grph*, *minWeight=-inf*, *maxWeight=inf*, *parameterName='weight'*, *ignoreUnweighted=False*, *dropSelfLoops=False*)

Modifies *grph* by dropping edges whose weight is not within the inclusive bounds of *minWeight* and *maxWeight*, i.e after running *grph* will only have edges whose weights meet the following inequality: *minWeight* <= edge's weight <= *maxWeight*. A `Keyerror` will be raised if the graph is unweighted unless *ignoreUnweighted* is `True`, the weight is determined by examining the attribute *parameterName*.

Note: none of the default options will result in *grph* being modified so only specify the relevant ones, e.g. `dropEdges(G, dropSelfLoops = True)` will remove only the self loops from `G`.

Parameters

grph : `networkx Graph`

The graph to be modified.

minWeight : `optional [int or double]`

default `-inf`, the minimum weight for an edge to be kept in the graph.

maxWeight : `optional [int or double]`

default `inf`, the maximum weight for an edge to be kept in the graph.

parameterName : `optional [str]`

default `'weight'`, key to weight field in the edge's attribute dictionary, the default is the same as networkx and metaknowledge so is likely to be correct

ignoreUnweighted : `optional [bool]`

default `False`, if `True` unweighted edges will kept

dropSelfLoops : `optional [bool]`

default `False`, if `True` self loops will be removed regardless of their weight

`metaknowledge.graphHelpers.`**`dropNodesByCount`**(*grph*, *minCount=-inf*, *maxCount=inf*, *parameterName='count'*, *ignoreMissing=False*)

Modifies *grph* by dropping nodes that do not have a count that is within inclusive bounds of *minCount* and *maxCount*, i.e after running *grph* will only have nodes whose degrees meet the following inequality: *minCount* <= node's degree <= *maxCount*.

Count is determined by the count attribute, *parameterName*, and if missing will result in a `KeyError` being raised. *ignoreMissing* can be set to `True` to suppress the error.

minCount and maxCount default to negative and positive infinity respectively so without specifying either the output should be the input

Parameters

grph : `networkx Graph`

The graph to be modified.

minCount : `optional [int or double]`

default `-inf`, the minimum Count for an node to be kept in the graph.

maxCount : `optional [int or double]`

default `inf`, the maximum Count for an node to be kept in the graph.

parameterName : `optional [str]`

default `'count'`, key to count field in the nodes's attribute dictionary, the default is the same throughout metaknowledge so is likely to be correct.

ignoreMissing : `optional [bool]`

default `False`, if `True` nodes missing a count will be kept in the graph instead of raising an exception

`metaknowledge.graphHelpers.`**`dropNodesByDegree`**(*grph*, *minDegree=-inf*, *maxDegree=inf*, *useWeight=True*, *parameterName='weight'*, *includeUnweighted=True*)

Modifies *grph* by dropping nodes that do not have a degree that is within inclusive bounds of *minDegree* and

maxDegree, i.e after running *grph* will only have nodes whose degrees meet the following inequality: *minDegree* <= node's degree <= *maxDegree*.

Degree is determined in two ways, the default *useWeight* is the weight attribute of the edges to a node will be summed, the attribute's name is *parameterName* otherwise the number of edges touching the node is used. If *includeUnweighted* is `True` then *useWeight* will assign a degree of 1 to unweighted edges.

Parameters

grph : `networkx Graph`

The graph to be modified.

minDegree : `optional [int or double]`

default `-inf`, the minimum degree for an node to be kept in the graph.

maxDegree : `optional [int or double]`

default `inf`, the maximum degree for an node to be kept in the graph.

useWeight : `optional [bool]`

default `True`, if `True` the the edge weights will be summed to get the degree, if `False` the number of edges will be used to determine the degree.

parameterName : `optional [str]`

default `'weight'`, key to weight field in the edge's attribute dictionary, the default is the same as networkx and metaknowledge so is likely to be correct.

includeUnweighted : `optional [bool]`

default `True`, if `True` edges with no weight will be considered to have a weight of 1, if `False` they will cause a `KeyError` to be raised.

metaknowledge.graphHelpers.**getNodeDegrees**(*grph*, *weightString='weight'*, *strictMode=False*, *returnType=<class 'int'>*, *edgeType='bi'*)

Retunrs a dictionary of nodes to their degrees, the degree is determined by adding the weight of edge with the weight being the string weightString that gives the name of the attribute of each edge containng thier weight. The Weights are then converted to the type returnType. If weightString is give as False instead each edge is counted as 1.

edgeType, takes in one of three strings: 'bi', 'in', 'out'. 'bi' means both nodes on the edge count it, 'out' mans only the one the edge comes form counts it and 'in' means only the node the edge goes to counts it. 'bi' is the default. Use only on directional graphs as otherwise the selected nodes is random.

metaknowledge.graphHelpers.**getWeight**(*grph*, *nd1*, *nd2*, *weightString='weight'*, *returnType=<class 'int'>*)

A way of getting the weight of an edge with or without weight as a parameter

returns a the value of the weight parameter converted to returnType if it is given or 1 (also converted) if not

metaknowledge.graphHelpers.**graphStats**(*G*, *stats=('nodes', 'edges', 'isolates', 'loops', 'density', 'transitivity')*, *makeString=True*, *sentenceString=False*)

Returns a string or list containing statistics about the graph *G*.

graphStats() gives 6 different statistics: number of nodes, number of edges, number of isolates, number of loops, density and transitivity. The ones wanted can be given to *stats*. By default a string giving each stat on

a different line it can also produce a sentence containing all the requested statistics or the raw values can be accessed instead by setting *makeString* to `False`.

Parameters

G : `networkx Graph`

The graph for the statistics to be determined of

stats : `optional [list or tuple [str]]`

Default `('nodes', 'edges', 'isolates', 'loops', 'density', 'transitivity')`, a list or tuple containing any number or combination of the strings:

`"nodes"`, `"edges"`, `"isolates"`, `"loops"`, `"density"` and ‘”transitivity”‘‘

At least one occurrence of the corresponding string causes the statistics to be provided in the string output. For the non-string (tuple) output the returned tuple has the same length as the input and each output is at the same index as the string that requested it, e.g.

```
_stats_ = ("edges", "loops", "edges")
```

The return is a tuple with 2 elements the first and last of which are the number of edges and the second is the number of loops

makeString : `optional [bool]`

Default `True`, if `True` a string is returned if `False` a tuple

sentenceString : `optional [bool]`

Default `False` : if `True` the returned string is a sentce, otherwise each value has a seperate line.

Returns

`str or tuple [float and int]`

The type is determined by *makeString* and the layout by *stats*

`metaknowledge.graphHelpers.`**`mergeGraphs`**(*targetGraph*, *addedGraph*, *incrementedNodeVal='count'*, *incrementedEdgeVal='weight'*)

A quick way of merging graphs, this is meant to be quick and is only intended for graphs generated by metaknowledge. This does not check anything and as such may cause unexpected results if the source and target were not generated by the same method.

mergeGraphs() will **modify** *targetGraph* in place by adding the nodes and edges found in the second, *addedGraph*. If a node or edge exists *targetGraph* is given precedence, but the edge and node attributes given by *incrementedNodeVal* and incrementedEdgeVal are added instead of being overwritten.

Parameters

targetGraph : `networkx Graph`

the graph to be modified, it has precedence.

addedGraph : `networkx Graph`

the graph that is unmodified, it is added and does **not** have precedence.

incrementedNodeVal : `optional [str]`

default `'count'`, the name of the count attribute for the graph's nodes. When merging this attribute will be the sum of the values in the input graphs, instead of *targetGraph*'s value.

incrementedEdgeVal : `optional [str]`

default `'weight'`, the name of the weight attribute for the graph's edges. When merging this attribute will be the sum of the values in the input graphs, instead of *targetGraph*'s value.

`metaknowledge.graphHelpers.`**`readGraph`**(*edgeList*, *nodeList=None*, *directed=False*, *idKey='ID'*, *eSource='From'*, *eDest='To'*)

Reads the files given by *edgeList* and *nodeList* and creates a networkx graph for the files.

This is designed only for the files produced by metaknowledge and is meant to be the reverse of *writeGraph()*, if this does not produce the desired results the networkx builtin networkx.read_edgelist() could be tried as it is aimed at a more general usage.

The read edge list format assumes the column named *eSource* (default `'From'`) is the source node, then the column *eDest* (default `'To'`) givens the destination and all other columns are attributes of the edges, e.g. weight.

The read node list format assumes the column *idKey* (default `'ID'`) is the ID of the node for the edge list and the resulting network. All other columns are considered attributes of the node, e.g. count.

Note: If the names of the columns do not match those given to **readGraph()** a `KeyError` exception will be raised.

Note: If nodes appear in the edgelist but not the nodeList they will be created silently with no attributes.

Parameters

edgeList : `str`

a string giving the path to the edge list file

nodeList : `optional [str]`

default `None`, a string giving the path to the node list file

directed : `optional [bool]`

default `False`, if `True` the produced network is directed from *eSource* to *eDest*

idKey : `optional [str]`

default `'ID'`, the name of the ID column in the node list

eSource : `optional [str]`

default `'From'`, the name of the source column in the edge list

eDest : `optional [str]`

default `'To'`, the name of the destination column in the edge list

Returns

`networkx Graph`

the graph described by the input files

`metaknowledge.graphHelpers.`**`writeEdgeList`**(*grph*, *name*, *extraInfo=True*, *allSameAttribute=False*, *_progBar=None*)

Writes an edge list of *grph* at the destination *name*.

The edge list has two columns for the source and destination of the edge, `'From'` and `'To'` respectively, then, if *edgeInfo* is `True`, for each attribute of the node another column is created.

Note: If any edges are missing an attribute it will be left blank by default, enable *allSameAttribute* to cause a `KeyError` to be raised.

Parameters

grph : `networkx Graph`

The graph to be written to *name*

name : `str`

The name of the file to be written

edgeInfo : `optional [bool]`

Default `True`, if `True` the attributes of each edge will be written

allSameAttribute : `optional [bool]`

Default `False`, if `True` all the edges must have the same attributes or an exception will be raised. If `False` the missing attributes will be left blank.

`metaknowledge.graphHelpers.`**`writeGraph`**(*grph*, *name*, *edgeInfo=True*, *typing=False*, *suffix='csv'*, *overwrite=True*, *allSameAttribute=False*)

Writes both the edge list and the node attribute list of *grph* to files starting with *name*.

The output files start with *name*, the file type (edgeList, nodeAttributes) then if typing is True the type of graph (directed or undirected) then the suffix, the default is as follows:

name_fileType.suffix

Both files are csv's with comma delimiters and double quote quoting characters. The edge list has two columns for the source and destination of the edge, `'From'` and `'To'` respectively, then, if *edgeInfo* is `True`, for each attribute of the node another column is created. The node list has one column call "ID" with the node ids used by networkx and all other columns are the node attributes.

To read back these files use *readGraph()* and to write only one type of lsit use *writeEdgeList()* or *writeNodeAttributeFile()*.

Warning: this function will overwrite files, if they are in the way of the output, to prevent this set *overwrite* to `False`

Note: If any nodes or edges are missing an attribute a `KeyError` will be raised.

Parameters

grph : `networkx Graph`

A networkx graph of the network to be written.

name : `str`

The start of the file name to be written, can include a path.

edgeInfo : `optional [bool]`

Default `True`, if `True` the the attributes of each edge are written to the edge list.

typing : `optional [bool]`

Default `False`, if `True` the directed ness of the graph will be added to the file names.

suffix : optional [str]

Default "csv", the suffix of the file.

overwrite : optional [bool]

Default True, if True files will be overwritten silently, otherwise an OSError exception will be raised.

metaknowledge.graphHelpers.**writeNodeAttributeFile**(*grph*, *name*, *allSameAttribute=False*, *_progBar=None*)

Writes a node attribute list of *grph* to the file given by the path *name*.

The node list has one column call 'ID' with the node ids used by networkx and all other columns are the node attributes.

Note: If any nodes are missing an attribute it will be left blank by default, enable *allSameAttribute* to cause a KeyError to be raised.

Parameters

grph : networkx Graph

The graph to be written to *name*

name : str

The name of the file to be written

allSameAttribute : optional [bool]

Default False, if True all the nodes must have the same attributes or an exception will be raised. If False the missing attributes will be left blank.

metaknowledge.graphHelpers.**writeTnetFile**(*grph*, *name*, *modeNameString*, *weighted=False*, *sourceMode=None*, *timeString=None*, *nodeIndexString='tnet-ID'*, *weightString='weight'*)

Writes an edge list designed for reading by the *R* package tnet.

The *networkx* graph provided must be a pure two-mode network, the modes must be 2 different values for the node attribute accessed by *modeNameString* and all edges must be between different node types. Each node will be given an integer id, stored in the attribute given by *nodeIndexString*, these ids are then written to the file as the endpoints of the edges. Unless *sourceMode* is given which mode is the source (first column) and which the target (second column) is random.

Note the *grph* will be modified by this function, the ids of the nodes will be written to the graph at the attribute *nodeIndexString*.

Parameters

grph : network Graph

The graph that will be written to *name*

name : str

The path of the file to write

modeNameString : str

The name of the attribute *grph*'s modes are stored in

weighted : optional bool

Default False, if True then the attribute *weightString* will be written to the weight column

sourceMode : optional str

Default None, if given the name of the mode used for the source (first column) in the output file

timeString : optional str

Default None, if present the attribute *timeString* of an edge will be written to the time column surrounded by double quotes (").

Note The format used by tnet for dates is very strict it uses the ISO format, down to the second and without time zones.

nodeIndexString : optional str

Default 'tnet-ID', the name of the attribute to save the id for each node

weightString : optional str

Default 'weight', the name of the weight attribute

Record is the base of various objects in mk, it is intended to be used with things that have some sort of key-value relationship and is basiclly a hashable python dict. It also has a few extra attributes intead to make debugging and record keeping easier.

- bad cand be set to True to indcate something is wrong with the issue being saved in error the exact details are left to designer
- _sourceFile and _sourceLine store the original file name and line number and are mostly for improving error messages
- _id should be a unique string, that preferably can be used to identify the record from its source, although the latter is not always possible to do so, do your best. It is also what is used for hashing and comparison
- _fieldDict contains the base mapping of keys to values, it is the dictionary

ExtendedRecord is what WOSRecord and its ilk inherit from and extends Record by adding memoizing and processing of the fields. ExtendedRecord cannot be invoked directly as it has many abstract (virtual) methods that define how the tags are to be proccesed what they are called, what encoding to use when writing to disk, etc.

metaknowledge.mkRecord.**_bibFormatter**(*s*, *maxLength*)

Formats a string, list or number to make it good for a bib file by:
* if too long splits up the string correctly
* tries to use the best quoting characters
* expands lists into ' and ' seperated values, as per spec for authors field
Note, this does not escape characters. LaTeX may have issues with the output
Max length splitting derived from https://www.cs.arizona.edu/~collberg/Teaching/07.231/BibTeX/bibtex.html

metaknowledge.recordCollection.**addToNetwork**(*grph*, *nds*, *count*, *weighted*, *nodeType*, *nodeInfo*, *fullInfo*, *coreCitesDict*, *coreValues*, *detailedValues*, *addCR*, *recordToCite=True*, *headNd=None*)

Addeds the citations *nds* to *grph*, according to the rules give by *nodeType*, *fullInfo*, etc.

headNd is the citation of the Record

metaknowledge.recordCollection.**expandRecs**(*G*, *RecCollect*, *nodeType*, *weighted*)

Expand all the citations from *RecCollect*

metaknowledge.recordCollection.**makeID**(*citation*, *nodeType*)
Makes the id, of the correct type for the network

metaknowledge.recordCollection.**makeNodeTuple**(*citation*, *idVal*, *nodeInfo*, *fullInfo*, *nodeType*, *count*, *coreCitesDict*, *coreValues*, *detailedValues*, *addCR*)
Makes a tuple of idVal and a dict of the selected attributes

metaknowledge.genders.nameGender.**nameStringGender**(*s*, *noExcept=False*)
Expects `first, last`

3.2.6 Exceptions

The exceptions defined by *metaknowledge* are:

exception metaknowledge.mkExceptions.**BadCitation**
Exception thrown by Citation

exception metaknowledge.mkExceptions.**BadGrant**

exception metaknowledge.mkExceptions.**BadInputFile**

exception metaknowledge.mkExceptions.**BadProQuestFile**

exception metaknowledge.mkExceptions.**BadProQuestRecord**

exception metaknowledge.mkExceptions.**BadPubmedFile**

exception metaknowledge.mkExceptions.**BadPubmedRecord**

exception metaknowledge.mkExceptions.**BadRecord**

exception metaknowledge.mkExceptions.**BadScopusFile**

exception metaknowledge.mkExceptions.**BadScopusRecord**

exception metaknowledge.mkExceptions.**BadWOSFile**
Exception thrown by wosParser for mis-formated files

exception metaknowledge.mkExceptions.**BadWOSRecord**
Exception thrown by the record parser to indicate a mis-formated record. This occurs when some component of the record does not parse. The messages will be any of:

```
* _Missing field on line (line Number):(line)_, which indicates a line was to␣
↪short, there should have been a tag followed by information

* _End of file reached before ER_, which indicates the file ended before the 'ER'␣
↪indicator appeared, 'ER' indicates the end of a record. This is often due to a␣
↪copy and paste error.

* _Duplicate tags in record_, which indicates the record had 2 or more lines with␣
↪the same tag.

* _Missing WOS number_, which indicates the record did not have a 'UT' tag.
```

Records with a BadWOSRecord error are likely incomplete or the combination of two or more single records.

exception metaknowledge.mkExceptions.**CollectionTypeError**

exception metaknowledge.mkExceptions.**GenderException**

exception metaknowledge.mkExceptions.**GrantCollectionException**

exception metaknowledge.mkExceptions.**JournalDataBaseError**

exception metaknowledge.mkExceptions.**RCTypeError**

exception metaknowledge.mkExceptions.**RCValueError**

exception metaknowledge.mkExceptions.**RecordsNotCompatible**

exception metaknowledge.mkExceptions.**TagError**

exception metaknowledge.mkExceptions.**UnknownFile**

exception metaknowledge.mkExceptions.**cacheError**

Exception raised when loading a cached RecordCollection fails, should only be seen inside metaknowledge and always be caught.

exception metaknowledge.mkExceptions.**mkException**

3.3 Examples

Note: for a more recent example of using *metaknowledge*, please visit the NetLab blog.

metaknowledge is a python library for creating and analyzing scientific metadata. It uses records obtained from Web of Science (WOS), Scopus and other sources. It is intended to be usable by those who do not know much python. This page will be a short overview of its capabilities, to allow you to use it for your own work.

This document was made from a jupyter notebook, if you know how to use them, you can download the notebook here and the sample file is here if you wish to have an interactive version of this page. Now let's begin.

3.3.1 About Jupyter Notebooks

This document was made from a jupyter notebook and can show and run python code. The document is broken up into what are called cells, each cell is either code, output, or markdown (text). For example this cell is markdown, which means it is plain text with a couple small formatting things, like the link in the first sentence. You can change the cell type using the dropdown menu at the top of the page.

[1]:
```
#This cell is python
#The cell below it is output
print("This is an output cell")
```

```
This is an output cell
```

The code cells contain python code that you can edit and run your self. Try changing the one above.

3.3.2 Importing

First you need to import the *metaknowledge* package

[2]:
```
import metaknowledge as mk
```

And you will often need the networkx package

[3]:
```
import networkx as nx
```

And matplotlib to display the graphs and to make them look nice when displayed

[4]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

metaknowledge also has a *matplotlib* based graph visualizer that will be used sometimes

[5]:
```
import metaknowledge.visual as mkv
```

These lines of code will be at the top of all the other lessons as they are what let us use *metaknowledge*.

3.3.3 Reading Files

First we need to import *metaknowledge* like we saw in lesson 1.

[1]:
```
import metaknowledge as mk
```

we only need *metaknowledge* for now so no need to import everything

The files from the Web of Science (WOS) can be loaded into a RecordCollections by creating a `RecordCollection` with the path to the files given to it as a string.

[2]:
```
RC = mk.RecordCollection("savedrecs.txt")
repr(RC)
```

[2]:
```
'savedrecs'
```

You can also read a whole directory, in this case it is reading the current working directory

[3]:
```
RC = mk.RecordCollection(".")
repr(RC)
```

[3]:
```
'files-from-.'
```

metaknowledge can detect if a file is a valid WOS file or not and will read the entire directory and load only those that have the right header. You can also tell it to only read a certain type of file, by using the extension argument.

[4]:
```
RC = mk.RecordCollection(".", extension = "txt")
repr(RC)
```

[4]:
```
'txt-files-from-.'
```

Now you have a `RecordCollection` composed of all the WOS records in the selected file(s).

[5]:
```
print("RC is a " + str(RC))
```

```
RC is a Collection of 32 records
```

You might have noticed I used two different ways to display the `RecordCollection`. `repr(RC)` will give you where *metaknowledge* thinks the collection came from. While `str(RC)` will give you a nice string containing the number of `Records`.

3.3.4 Objects

In *Python* everything is an object thus everything *metaknowledge* produces is an object. There are three objects that have been created specifically for it, objects created this way are call classes. The three are `Record` a single WOS record, `RecordCollection` a group of `Records` and `Citation` a single WOS citation.

Lets import *metaknowledge* and read a file

[1]:
```
import metaknowledge as mk
RC = mk.RecordCollection('../savedrecs.txt') # '..' is one directory above the␣
↪current one
```

Now we can look at how the different objects relate to this file.

3.3.5 `Record` object

Record is an object that contains a simple WOS record, for example a journal article, book, or conference proceedings. They are what RecordCollections contain. To see an individual Record at random from a `RecordCollection` you can use `peak()`

[2]:
```
R = RC.peak()
```

A single `Record` can give you all the information it contains about its record. If for example you want its authors.

[3]:
```
print(R.authorsFull)
print(R.AF)
```

```
['BREVIK, I']
['BREVIK, I']
```

Converting a `Record` to a string will give its title

[4]:
```
print(R)
```

```
EXPERIMENTS IN PHENOMENOLOGICAL ELECTRODYNAMICS AND THE ELECTROMAGNETIC␣
↪ENERGY-MOMENTUM TENSOR
```

If you try to access a tag the `Record` does not have it will return `None`

[5]:
```
print(R.GP)
```

```
None
```

There are two ways of getting each tag, one is using the WOS 2 letter abbreviation and the second is to use the human readable name. There is no standard for the human readable names, so they are specific to *metaknowledge*. To see how the WOS names map to the long names look at tagFuncs. If you want all the tags a `Record` has use iter.

[6]:
```
print(R.__iter__())
```

```
['PT', 'AU', 'AF', 'TI', 'SO', 'LA', 'DT', 'C1', 'CR', 'NR', 'TC', 'Z9', 'PU', 'PI',␣
↪'PA', 'SN', 'J9', 'JI', 'PY', 'VL', 'IS', 'BP', 'EP', 'DI', 'PG', 'WC', 'SC', 'GA',␣
↪'UT']
```

3.3.6 `RecordCollection` object

RecordCollection is the object that *metaknowledge* uses the most. It is your interface with the data you want.

To iterate over all of the `Records` you can use a for loop

[7]:
```
for R in RC:
    print(R)
```

```
EXPERIMENTS IN PHENOMENOLOGICAL ELECTRODYNAMICS AND THE ELECTROMAGNETIC␣
↪ENERGY-MOMENTUM TENSOR
OBSERVATION OF SHIFTS IN TOTAL REFLECTION OF A LIGHT-BEAM BY A MULTILAYERED STRUCTURE
ANGULAR SPECTRUM AS AN ELECTRICAL NETWORK
SHIFTS OF COHERENT-LIGHT BEAMS ON REFLECTION AT PLANE INTERFACES BETWEEN ISOTROPIC␣
↪MEDIA
DISCUSSIONS OF PROBLEM OF PONDEROMOTIVE FORCES
A Novel Method for Enhancing Goos-Hanchen Shift in Total Internal Reflection
Optical properties of nanostructured thin films
Simple technique for measuring the Goos-Hanchen effect with polarization modulation␣
↪and a position-sensitive detector
CONSERVATION OF ANGULAR MOMENT WITH SIX COMPONENTS AND ASYMMETRICAL IMPULSE ENERGY␣
↪TENSORS
INTERFERENCE THEORY OF REFLECTION FROM MULTILAYERED MEDIA
Longitudinal and transverse effects of nonspecular reflection
TRANSVERSE DISPLACEMENT OF A TOTALLY REFLECTED LIGHT-BEAM AND PHASE-SHIFT METHOD
MECHANICAL INTERPRETATION OF SHIFTS IN TOTAL REFLECTION OF SPINNING PARTICLES
WHY ENERGY FLUX AND ABRAHAMS PHOTON MOMENTUM ARE MACROSCOPICALLY SUBSTITUTED FOR␣
↪MOMENTUM DENSITY AND MINKOWSKIS PHOTON MOMENTUM
SPIN ANGULAR-MOMENTUM OF A FIELD INTERACTING WITH A PLANE INTERFACE
Numerical study of the displacement of a three-dimensional Gaussian beam transmitted␣
↪at total internal reflection. Near-field applications
LONGITUDINAL AND TRANSVERSE DISPLACEMENTS OF A BOUNDED MICROWAVE BEAM AT TOTAL␣
↪INTERNAL-REFLECTION
EXCHANGED MOMENTUM BETWEEN MOVING ATOMS AND A SURFACE-WAVE - THEORY AND EXPERIMENT
ASYMMETRICAL MOMENTUM-ENERGY TENSORS AND 6-COMPONENT ANGULAR-MOMENTUM IN PROBLEM␣
↪CONCERNING 2 PHOTON MOMENTA AND MAGNETODYNAMIC EFFECT PROBLEM
Experimental observation of the Imbert-Fedorov transverse displacement after a single␣
↪total reflection
RESONANCE EFFECTS ON TOTAL INTERNAL-REFLECTION AND LATERAL (GOOS-HANCHEN) BEAM␣
↪DISPLACEMENT AT THE INTERFACE BETWEEN NONLOCAL AND LOCAL DIELECTRIC
Goos-Hanchen shift as a probe in evanescent slab waveguide sensors
THEORETICAL NOTES ON AMPLIFICATION OF TRANSVERSE SHIFT BY TOTAL REFLECTION ON␣
↪MULTILAYERED SYSTEM
INTERNAL PHOTON IMPULSE OF DIELECTRIC AND ON COUPLE APPLIED TO ANISOTROPIC CRYSTAL
SPIN ANGULAR-MOMENTUM OF A FIELD INTERACTING WITH A PLANE INTERFACE
CALCULATION AND MEASUREMENT OF FORCES AND TORQUES APPLIED TO UNIAXIAL CRYSTAL BY␣
↪EXTRAORDINARY WAVE
Goos-Hanchen and Imbert-Fedorov shifts for leaky guided modes
PREDICTION OF A RESONANCE-ENHANCED LASER-BEAM DISPLACEMENT AT TOTAL␣
↪INTERNAL-REFLECTION IN SEMICONDUCTORS
GENERAL STUDY OF DISPLACEMENTS AT TOTAL REFLECTION
NONLINEAR TOTALLY REFLECTING PRISM COUPLER - THERMOMECHANIC EFFECTS AND␣
↪INTENSITY-DEPENDENT REFRACTIVE-INDEX OF THIN-FILMS
DISPLACEMENT OF A TOTALLY REFLECTED LIGHT-BEAM - FILTERING OF POLARIZATION STATES AND␣
↪AMPLIFICATION
Transverse displacement at total reflection near the grazing angle: a way to␣
↪discriminate between theories
```

The individual `Records` are index by their WOS numbers so you can access a specific one in the collection if you know its number.

[8]:
```
RC.getWOS("WOS:A1979GV55600001")
```

[8]:
```
<metaknowledge.record.Record at 0x7f07784be860>
```

3.3.7 `Citation` object

Citation is an object to contain the results of parsing a citation. They can be created from a `Record`

[9]:
```
Cite = R.createCitation()
print(Cite)
```

```
Pillon F, 2005, APPL PHYS B-LASERS O, V80, P355, DOI 10.1007/s00340-005-1728-2
```

`Citations` allow for the raw strings of citations to be manipulated easily by *metaknowledge*.

3.3.8 Filtering

The for loop shown above is the main way to filter a RecordCollection, that said there are a few builtin filters, e.g. yearSplit(), but the for loop is an easily generalized way of filtering that is relatively simple to read so it the main way you should filter. An example of the workflow is as follows:

First create a new RecordCollection

[10]:
```
RCfiltered = mk.RecordCollection()
```

Then add the records that meet your condition, in this case that their title's start with `'A'`

[11]:
```
for R in RC:
    if R.title[0] == 'A':
        RCfiltered.addRec(R)
```

[12]:
```
print(RCfiltered)
```

```
Collection of 3 records
```

Now you have a RecordCollection `RCfiltered` of all the `Records` whose titles begin with `'A'`.

One note about implementing this, the above code does not handle the case in which the title is missing i.e. `R.title` is `None`. You will have to deal with this on your own.

Two builtin functions to filter collections are yearSplit() and localCitesOf(). To get a RecordCollection of all Records between 1970 and 1979:

[13]:
```
RC70 = RC.yearSplit(1970, 1979)
print(RC70)
```

```
Collection of 19 records
```

The second function localCitesOf() takes in an object that a Citation can be created from and returns a RecordCollection of all the Records that cite it. So to see all the records that cite `"Yariv A., 1971, INTRO OPTICAL ELECTR"`.

[14]:
```
RCintroOpt = RC.localCitesOf("Yariv A., 1971, INTRO OPTICAL ELECTR")
print(RCintroOpt)
```

```
Collection of 1 records
```

3.3.9 Exporting RecordCollections

Now you have a filtered RecordCollection you can write it as a file with writeFile()

[15]:
```
RCfiltered.writeFile("Records_Starting_with_A.txt")
```

The written file is identical to one of those produced by WOS.

If you wish to have a more useful file use writeCSV() which creates a CSV file of all the tags as columns and the Records as rows. IF you only care about a few tags the `onlyTheseTags` argument allows you to control the tags.

[16]:
```
selectedTags = ['TI', 'UT', 'CR', 'AF']
```

This will give only the title, WOS number, citations, and authors.

[17]:
```
RCfiltered.writeCSV("Records_Starting_with_A.csv", onlyTheseTags = selectedTags)
```

The last export feature is for using *metaknowledge* with other packages, in particular pandas, which you will learn about later, but others should also work. makeDict() creates a dictionary with tags as keys and lists as values with each index of the lists corresponding to a Record. *pandas* can accept these directly to make DataFrames.

[18]:
```
import pandas
recDataFrame = pandas.DataFrame(RC.makeDict())
```

3.3.10 Making a network

For this class most of the types of network you will want to make can be produced by *metaknowledge*. The first three co-citation network, citation network and co-author network are specialized versions of the last three one-mode network, two-mode network and multi-mode network.

First we need to import metaknowledge and because we will be dealing with graphs the graphs package *networkx* as should be imported

[1]:
```
import metaknowledge as mk
import networkx as nx
```

And so we can visualize the graphs

[2]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import metaknowledge.contour.plotting as mkv
```

Before we start we should also get a `RecordCollection` to work with.

[3]:
```
RC = mk.RecordCollection('../savedrecs.txt')
```

Now lets look at the different types of graph.

3.3.11 Making a co-citation network

To make a basic co-citation network of Records use networkCoCitation().

[4]:
```
CoCitation = RC.networkCoCitation()
print(mk.graphStats(CoCitation, makeString = True)) #makestring by default is True so␣
↪it is not strictly necessary to include
```

```
The graph has 601 nodes, 19492 edges, 0 isolates, 4 self loops, a density of 0.108109␣
↪and a transitivity of 0.691662
```

graphStats() is a function to extract some of the statists of a graph and make them into a nice string.

`CoCitation` is now a networkx graph of the co-citation network, with the hashes of the `Citations` as nodes and the full citations stored as an attributes. Lets look at one node

[5]:
```
CoCitation.nodes(data = True)[0]
```

[5]:
```
(5308678917494226943,
 {'count': 1, 'info': 'CAVALLERI G, 1974, LETT NUOVO CIMENTO, V12, P626'})
```

and an edge

[6]:
```
CoCitation.edges(data = True)[0]
```

[6]:
```
(5308678917494226943, 7204849785423671553, {'weight': 1})
```

All the graphs *metaknowledge* use are *networkx* graphs, a few functions to trim them are implemented in *metaknowledge*, *here* is the example section, but many useful functions are implemented by it. Read the documentation here for more information.

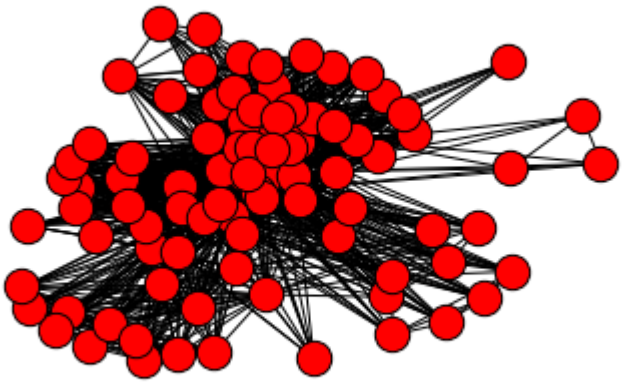
The `networkCoCitation()` function has many options for filtering and determining the nodes. The default is to use the `Citations` themselves. If you wanted to make a network of co-citations of journals you would have to make the node type `'journal'` and remove the non-journals.

[7]:
```
coCiteJournals = RC.networkCoCitation(nodeType = 'journal', dropNonJournals = True)
print(mk.graphStats(coCiteJournals))
```

```
The graph has 89 nodes, 1383 edges, 0 isolates, 40 self loops, a density of 0.353166␣
↪and a transitivity of 0.640306
```

Lets take a look at the graph after a quick spring layout

[8]:
```
nx.draw_spring(coCiteJournals)
```



A bit basic but gives a general idea. If you want to make a much better looking and more informative visualization you could try gephi or visone. Exporting to them is covered below in *Exporting graphs*.

3.3.12 Making a citation network

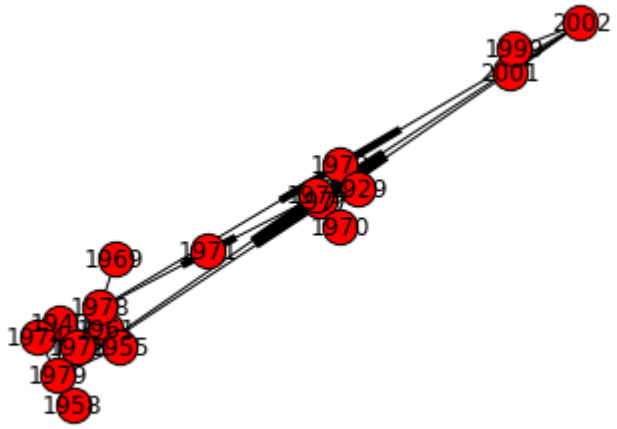
The networkCitation() method is nearly identical to `networkCoCitation()` in its parameters. It has one additional keyword argument `directed` that controls if it produces a directed network. Read *Making a co-citation network* to learn more about `networkCitation()`.

One small example is still worth providing. If you want to make a network of the citations of years by other years and have the letter `'A'` in them then you would write:

[9]:
```
citationsA = RC.networkCitation(nodeType = 'year', keyWords = ['A'])
print(mk.graphStats(citationsA))
```

```
The graph has 18 nodes, 24 edges, 0 isolates, 1 self loops, a density of 0.0784314␣
↪and a transitivity of 0.0344828
```

[10]:
```
nx.draw_spring(citationsA, with_labels = True)
```



3.3.13 Making a co-author network

The networkCoAuthor() function produces the co-authorship network of the RecordCollection as is used as shown

[11]:
```
coAuths = RC.networkCoAuthor()
print(mk.graphStats(coAuths))
```

```
The graph has 45 nodes, 46 edges, 9 isolates, 0 self loops, a density of 0.0464646␣
↪and a transitivity of 0.822581
```

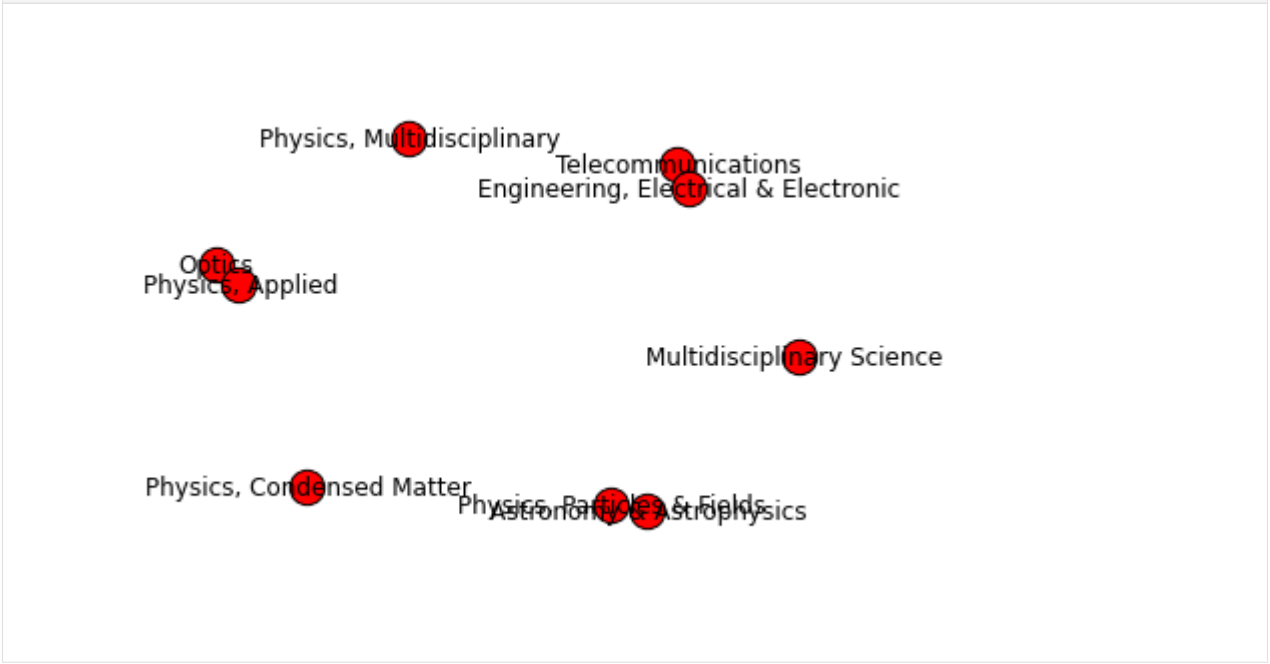
3.3.14 Making a one-mode network

In addition to the specialized network generators *metaknowledge* lets you make a one-mode co-occurence network of any of the WOS tags, with the oneModeNetwork() function. For examples the WOS subject tag `'WC'` can be examined.

[12]:
```
wcCoOccurs = RC.oneModeNetwork('WC')
print(mk.graphStats(wcCoOccurs))
```

```
The graph has 9 nodes, 3 edges, 3 isolates, 0 self loops, a density of 0.0833333 and␣
↪a transitivity of 0
```

[13]:
```
nx.draw_spring(wcCoOccurs, with_labels = True)
```



3.3.15 Making a two-mode network

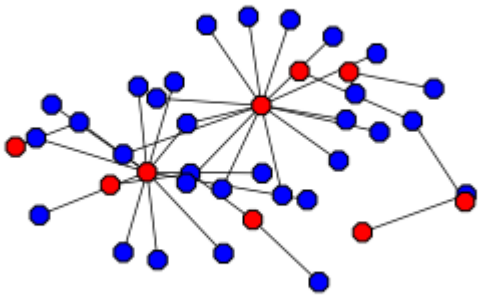
If you wish to study the relationships between 2 tags you can use the twoModeNetwork() function which creates a two mode network showing the connections between the tags. For example to look at the connections between titles(`'TI'`) and subjects (`'WC'`)

[14]:
```
ti_wc = RC.twoModeNetwork('WC', 'title')
print(mk.graphStats(ti_wc))
```

```
The graph has 40 nodes, 35 edges, 0 isolates, 0 self loops, a density of 0.0448718␣
↪and a transitivity of 0
```

The network is directed by default with the first tag going to the second.

[15]:
```
mkv.quickVisual(ti_wc, showLabel = False) #default is False as there are usually lots␣
↪of labels
```



quickVisual() makes a graph with the different types of nodes coloured differently and a couple other small visual tweaks from *networkx*'s `draw_spring`.

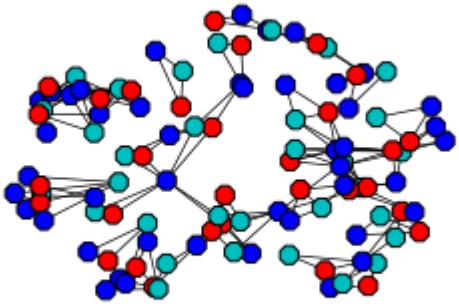
3.3.16 Making a multi-mode network

For any number of tags the nModeNetwork() function will do the same thing as the `oneModeNetwork()` but with any number of tags and it will keep track of their types. So to look at the co-occurence of titles `'TI'`, WOS number `'UT'` and authors `'AU'`.

[16]:
```
tags = ['TI', 'UT', 'AU']
multiModeNet = RC.nModeNetwork(tags)
mk.graphStats(multiModeNet)
```

[16]:
```
'The graph has 108 nodes, 163 edges, 0 isolates, 0 self loops, a density of 0.0282105␣
↪and a transitivity of 0.443946'
```

[17]:
```
mkv.quickVisual(multiModeNet)
```

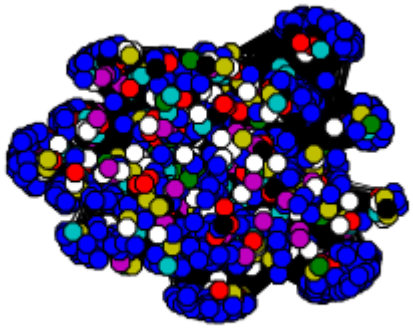


Beware this can very easily produce hairballs

[18]:
```
tags = mk.tagsAndNames #All the tags, twice
sillyMultiModeNet = RC.nModeNetwork(tags)
mk.graphStats(sillyMultiModeNet)
```

[18]:
```
'The graph has 1184 nodes, 59573 edges, 0 isolates, 1184 self loops, a density of 0.
↪0850635 and a transitivity of 0.492152'
```

[19]:
```
mkv.quickVisual(sillyMultiModeNet)
```



3.3.17 Post processing graphs

If you wish to apply a well known algorithm or process to a graph networkx is a good place to look as they do a good job at implementing them.

One of the features it lacks though is pruning of graphs, *metaknowledge* has these capabilities. To remove edges outside of some weight range, use dropEdges(). For example if you wish to remove the self loops, edges with weight less than 2 and weight higher than 10 from `coCiteJournals`.

[20]:
```
minWeight = 3
maxWeight = 10
proccessedCoCiteJournals = mk.dropEedges(coCiteJournals, minWeight, maxWeight,␣
↪dropSelfLoops = True)
mk.graphStats(proccessedCoCiteJournals)
```

[20]:
```
'The graph has 89 nodes, 466 edges, 1 isolates, 0 self loops, a density of 0.118999␣
↪and a transitivity of 0.213403'
```

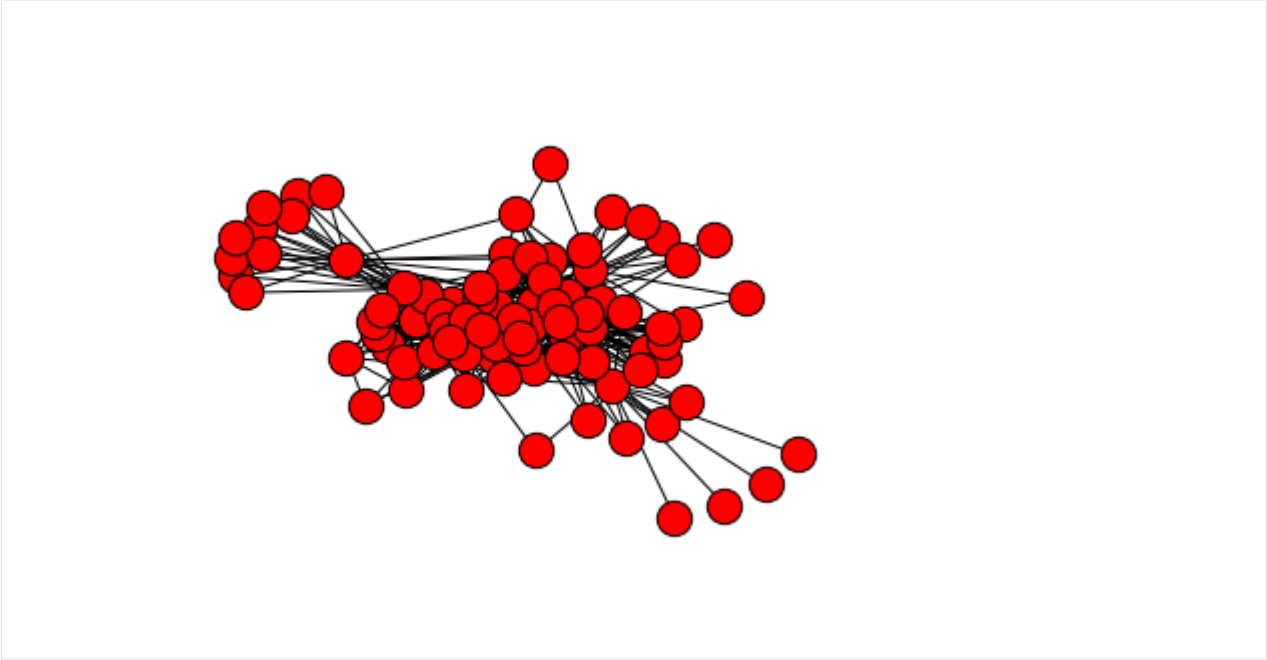
Then to remove all the isolates, i.e. nodes with degree less than 1, use dropNodesByDegree()

[21]:
```
proccessedCoCiteJournals = mk.dropNodesByDegree(proccessedCoCiteJournals, 1)
mk.graphStats(proccessedCoCiteJournals)
```

[21]:
```
'The graph has 88 nodes, 466 edges, 0 isolates, 0 self loops, a density of 0.121735␣
↪and a transitivity of 0.213403'
```

Now before the processing the graph can be seen *here*. After the processing it looks like

[22]:
```
nx.draw_spring(proccessedCoCiteJournals)
```



Hm, it looks a bit thinner. Using a visualizer will make the difference a bit more noticeable.

3.3.18 Exporting graphs

Now you have a graph the last step is to write it to disk. *networkx* has a few ways of doing this, but they tend to be slow. *metaknowledge* can write an edge list and node attribute file that contain all the information of the graph. The function to do this is called writeGraph(). You give it the start of the file name and it makes two labeled files containing the graph.

[23]:
```
mk.writeGraph(proccessedCoCiteJournals, "FinalJournalCoCites")
```

These files are simple CSVs an can be read easily by most systems. If you want to read them back into Python the readGraph() function will do that.

[24]:
```
FinalJournalCoCites = mk.readGraph("FinalJournalCoCites_edgeList.csv",
↪"FinalJournalCoCites_nodeAttributes.csv")
mk.graphStats(FinalJournalCoCites)
```

[24]:
```
'The graph has 88 nodes, 466 edges, 0 isolates, 0 self loops, a density of 0.121735␣
↪and a transitivity of 0.213403'
```

This is full example workflow for *metaknowledge*, the package is flexible and you hopefully will be able to customize it to do what you want (I assume you do not want the Records staring with 'A').

3.4 Command Line Tool

metaknowledge comes with a command-line application named `metaknowledge`. This provides a simple interface to the python package and allows the generation of most of the networks along with ways to manage the records themselves.

3.4.1 Overview

To start the tool run:

```
$ metaknowledge
```

You will be asked for the location of the file or files to use. These can be given by paths to the files or paths to directories with the files. Note: if a directory is used all files with the proper header will be read.

You will then be asked what to do with the records:

```
A collection of 537 WOS records has been created
What do you wish to do with it:
1) Make a graph
2) Write the collection as a single WOS style file
3) Write the collection as a single WOS style file and make a graph
4) Write the collection as a single csv file
5) Write the collection as a single csv file and make a graph
6) Write all the citations to a single file
7) Go over non-journal citations
i) open python console
q) quit
What is your selection:
```

Select the option you want by typing the corresponding number or character and pressing enter. The menus after this step are controlled this way as well.

The second last option `i)` will start an interactive python session will all the objects you have created thus far accessible, their names will be given when it starts.

The last option `q)` will cause the program to exit. You can also quit at any time by pressing `ctr-c`.

3.4.2 Questions?

If you find bugs, or have questions, please write to:

Reid McIlroy-Young reid@reidmcy.com
John McLevey john.mclevey@uwaterloo.ca

3.4.3 License

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

B

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y