pacer_lib Documentation

Release 2.33

Charles Zhang and Kevin Jiang

June 17, 2015

Contents

pacer_lib was made possible and is maintained by the Coase-Sandor Institute for Law and Economics at the University of Chicago Law School.

CHAPTER 1

Overview

pacer_lib is a library that has been designed to facilitate empirical legal research that uses data from the Public Access to Court Electronic Records (PACER) database by providing easy-to-use objects for scraping, parsing and searching PACER docket sheets and documents.

We developed *pacer_lib* in order to solve problems that arose naturally during the course of our research and our goal is to make it easy to:

- **download a large number of specific documents from PACER**

  To locate and download multiple files on PACER requires a lot of manual labour, so one of the first things that we developed was a way to programatically interface with the PACER Case Locator so that we could script all of our docket and document requests.

- **store downloaded dockets in a sensible and scalable way**

  PACER charges a fee for every page and document you access. If you have a project of any reasonable size and limited means, it becomes extremely important to keep track of what files you have already downloaded (lest you inadvertently download the file twice). We create a well-documented folder structure and a equally well-documented unique identifier that can be quickly disaggregated into a PACER search query or PACER case number.

- **extract information and create datasets from these dockets**

  We needed to create datasets for regression and textual analysis, so we baked in the process of converting relatively unstructured data (.html docket sheets) into more structured data (.csv for docket sheets and .json objects for other meta information).

CHAPTER 2

Table of Contents

2.1 ...Before You Begin

1. Please note that you will have to register for your own PACER account before you can download any documents from PACER (case-search only account is sufficient). The creators and maintainers of *pacer_lib* are in no way responsible for any charges you may incur on the PACER website as a result of using *pacer_lib*. You are also responsible for making sure that your use of *pacer_lib* complies with PACER's terms of service. **This library is provided as-is and you use it at your own risk.**
2. If you are looking for alternatives to accessing PACER directly, you could consider using RECAP or Bloomberg Law. You may be able to find ways to use *pacer_lib* to organize and parse output from these alternatives, but we do not provide any support for output from these systems as we haven't used these systems ourselves.

2.2 Installation

2.2.1 First Install

pacer_lib is published on PyPi, the standard Python module repository, While you can download the tarball from the website, we suggest that you use setuptools to install *pacer_lib*.

Specifically, if you have either `easy_install` or `pip` installed, just type:

```
pip install pacer_lib
```

or

```
easy_install pacer_lib
```

2.2.2 Compatibility and Required Libraries

In case you are running into trouble using *pacer_lib* or are looking to develop or modify *pacer_lib*, we have provided some notes on the system on which we developed *pacer_lib*.

We developed *pacer_lib* on Cygwin x86 in Windows 7 for Python 2.7.6.

We make extensive use of Requests v2.3.0 and BeautifulSoup 4. We also use datetime and, in older versions of *pacer_lib*, we use lxml.

2.3 Tutorial

We can't document every single use-case, but in this section, we will show code examples for some common (in our mind, at least) tasks to give you an idea of how to use *pacer_lib*. In addition, we'll make note of lower-level functions that you can also access in case you need more customized functionality.

2.3.1 1. Downloading Dockets

Code example

Downloading one document:

```
from pacer_lib.scraper import search_agent()

court_id = 'almdce'
case_number = '2:00-cv-00084'

# Login
s = search_agent("[Username]", "[Password]")

# Download
s.download_case_docket(case_number,court_id)
```

Downloading Multiple Documents:

```
from pacer_lib.scraper import search_agent()

# Login the search_agent into PACER
s = search_agent("[Username]", "[Password]")

cases = [('almdce', '2:00-cv-00084'),
         ('azdce', '2:98-cv-00020'),
         ('cacdce', '2:95-cv-03731')]

# Download the case dockets to ./results/local_docket_archive/
for court_id, case_num in cases:
    s.download_case_docket(case_num, court_id)
```

As you can see, before you begin you will need:

- a valid PACER username and password
- court ids and case numbers in a PACER case-number format for cases that you want to download

PACER username and password

You'll need to get that here.

For most purposes, you will register for a "PACER - Case Search Only Registration"

Court Id

This is an identifier for the court that you are searching. Usually, this is not particularly difficult to figure out.

For example, the court id **almdce** is made up of three parts:

- **al** – the state abbreviation for "Alabama"
- **md** – the abbreviation for "Middle District"
- **ce** – not sure what this stands for, but it's what PACER wants

For the equivalent bankruptcy court, **md** (Middle District) is changed to **mb** (Middle District Bankruptcy). If the state only has a single district, then the abbreviation is just **d**.

For example, the Massachusetts district court's court id is **madce**.

Appellate courts and the Supreme Court have not been implemented yet in *pacer_lib* yet.

To see a listing of all of the courts on PACER, you can go to this page.

PACER Case-Numbers

If you login to the PACER Case Locator, they will tell you that any of these formats can be used:

- yy-nnnnn
- yy-tp-nnnnn
- yy tp nnnnn
- yytpnnnnn
- o:yy-nnnnn
- o:yy-tp-nnnnn
- o:yy tp nnnnn
- o:yytpnnnnn

where:

- *yy* is the case-year (2 or 4 digits)
- *nnnnn* is the case-number (up to 5 digits)
- *tp* is the case type (e.g., 'cv', 'cr', 'bk', etc.)
- *o* is the office where the case was filed (1 digit)

pacer_lib works best with the clearest and mostly heavily delimited version:

- **o:yy-tp-nnnnn**

We use 2-digit years and we appended leading zeros to the *nnnnn* section if the case-number is less than 5 digits long.

Downloaded Filename

Files downloaded by `scraper.search_agent.download_case_docket()` are saved in the format: (court_id)_(case_num).html with colons replaced by plus signs, e.g., ('almdce', '2:00-cv-00084') is saved as 'almdce_2+00-cv-00084.html'.

Advanced Usage

For more information, look at the documentation at the object and function reference for `pacer_lib.scraper`. Here are some suggestions about how to do more complicated docket downloading:

- If you want to make your own searches you can use `search_agent.search_case_locator()` to create your own searches with other parameters.
- Once you have created your own searches and determined which dockets you want to download, you can use `search_agent.request_docket_sheet()` to download the docket.
- If you need to craft your own POST request, you can code it yourself using Requests or use `search_agent.query_case_locator()`.

If you would like to create your own POST request and pass them to

2.3.2 2. Parsing Downloaded Dockets

Code example

We are normally interested in parsing an entire directory of dockets at once (an this has minimal costs as all of the dockets are already local):

```
from pacer_lib.reader import docket_parser

# initialize a default docket_parser() object
# the default values look for dockets in './results/local_docket_archive/'
# and outputs to './results/processed_dockets/'
p = docket_parser()

# extract all docket information and case meta from dockets in the input
# directory and save the data to the output directory
p.parse_dir()
```

It is generally a bit unusual to just parse one file and you can always just parse the entire directory and find the parsed afterwards, but to prove that we can:

```
from pacer_lib.reader import docket_parser

# initialize a default docket_parser() object
# the default values look for dockets in
# './results/local_docket_archive/'
# and outputs to './results/processed_dockets/'

p = docket_parser()

# open a file, parse the file

file = './results/local_docket_archive/almdce_2+00-cv-00084.html'
with open(file, 'r') as f:
    print p.parse_data(f.read())
    print p.extract_all_meta(f.read())
```

Default Directories

`reader.docket_parser.parse_dir()` will output to the default output directory. Unless otherwise specified, the output directory will be './results/'. Within this output directory, there will be two sub directories created:

- `/parsed_dockets/`

  contains .csv documents that correspond to specific dockets

- **`/parsed_dockets_meta/` which contains two additional directories:**

- `/case_meta/`

  case_meta refers to the header information about the docket entries, e.g., assigned judge, case name, jurisdiction, etc. It also includes information about the lawyers who are associated with the case.

- `/download_meta/`

  download_meta refers to the information about the case that can be found on the PACER Case Locator results page. It also records when the docket was downloaded (only in newer versions).

Notes

- In older versions of *pacer_lib* (<= v2.32), we used `/processed_dockets/` and `/processed_dockets_meta/` as the default folders for `docket_parser`.

2.3.3 3. Searching Parsed Dockets

Code example

Example 1: After parsing all of the dockets using docket_parser, search for documents that are described with the word "judge" and "dismiss" but that does not include the word "foreign":

```
from pacer_lib.reader import docket_processor

r = docket_processor()
r.search_dir(require_term=['judge', 'dismiss'],
             exclude_term=['foreign'])
r.write_all_matches('firstsearch')
```

In this code example, all document entries that match this criteria will be written into a single file called 'all_match_firstsearch.csv'.

Example 2: Alternatively, search for the word "motion" in the first 10 characters of a document description and then write a result file for each case docket:

```
from pacer_lib.reader import docket_processor

r = docket_processor()
r.search_dir(require_term=['motion'],
             within='10')
r.write_individual_matches('motion10')
```

In this code example, all document entries from a single case will be written into a corresponding case file in a folder called '/docket_hits/' in the output path.

For example, if the case *(almdce, 2:00-cv-00084)* has 3 documents that have the word "motion" in the first 10 characters of their document description, then those 3 document entries will be written a new file called '^almdce_2+00-cv-00084_motion10.csv'.

Advanced Usage

The function `reader.docket_processor.search_dir()` commits its search results to the `reader.docket_processor.hit_list` variable inclusively. This means that you can run

`reader.docket_processor.search_dir()` several times if you want to simulate an **OR** boolean search:

```
from pacer_lib.reader import docket_processor

r = docket_processor()
r.search_dir(require_term=['motion'],
             within='10')
r.search_dir(require_term=['opinion'],
             within='10')
r.write_individual_matches('motion10')
```

AND searches and **NOT** searches, obviously, are built into the `require_term` and `exclude_term` arguments.

2.3.4 4. Downloading Documents

Code example

After parsing a docket, you can downloading a single document very simply:

```
from pacer_lib.scraper import search_agent()

# Document information, can be taken from parsed csv
case_filename = 'almdce_2+00-cv-00084'
doc_no = '31'
doc_link = 'https://ecf.almd.uscourts.gov/doc1/017149132'


# Login
s = search_agent("[Username]", "[Password]")

# Download
s.download_document(case_filename, doc_no, doc_link)
```

Advanced Usage

The actual document request and its raw response data (binary) can also be exposed using the `scraper.search_agent.request_document()` function.

2.3.5 5. Sorting Documents

Code example

This code hasn't been implemented yet.

2.4 pacer_lib.scraper

2.4.1 search_agent

class `pacer_lib.scraper.`**`search_agent`**(*username*, *password*, *output_path='./results'*, *auto_login=True*, *wait_time=1*)

Returns a `search_agent()` object, that serves as an interface for the PACER case locator. It will query and

download both dockets and documents. It is a modified requests.sessions object.

Keyword Arguments

- `username`: a valid PACER username
- `password`: a valid PACER password that goes with `username`
- `output_path`: allows you to specify the relative path where you would like to save your downloads. The actual docket sheets will be saved to a subfolder within output_path, '/local_docket_archive/'. If the folders do not exist, they will be created.
- `auto_login`: specify if you would like to login when the object is instantiated (you may want to use `search_agent()` to create PACER query strings).
- `wait_time`: how long to wait between requests to the PACER website.

download_case_docket(*case_no*, *court_id*, *other_options={'default_form': 'b', 'court_type': 'all'}*, *overwrite=False*)

Returns a list that indicates the case_no, court_id and any error. `download_case_docket` also writes the .html docket sheet to *self.output_path* (in a subfolder '/local_docket_archive/'. If you set *overwrite*=True, it will overwrite previous dockets. Otherwise, ''download_case_docket'' will check to see if the docket has already been downloaded **before** incurring any additional search or download charges.

You can also pass additional POST requests through *other_options*.

download_document(*case_filename*, *doc_no*, *doc_link*, *no_type='U'*, *overwrite=False*)

Returns a list that indicates the case_name, doc_no and any error. `download_case_document` also writes the .pdf document to *self.output_path* (to the sub-folder '/local_document_archive/'. If you set *overwrite*=True, it will overwrite previously downloaded documents. Otherwise, ''download_case_document'' will check to see if the docket has already been downloaded **before** incurring any additional search or download charges.

(To be implemented) docket_parser() assigns two types of numbers: the listed docket number (i.e., the number listed on the page) and the unique identifier (i.e., the position of the docket entry on the page). We should default to using the unique identifier, but all of the legacy files will be using the listed identifier and we will need to reassociate / convert those documents to their unique identifier.

no_type = 'U' –> unique identifier no_type = 'L' –> listed identifier

We have begun implementing this, but this is not completely finished.

Using the listed identifier should be considered legacy and not advised.

This will be dangerous in terms of redundant download protection.

Document this properly once we finish.

(Not implemented) You can also pass additional POST requests through *other_options*.

query_case_locator(*payload*)

Returns a string literal of the HTML of the search results page. This function passes queries to the PACER Case Locator (https://pcl.uscourts.gov/dquery) and this is the simplest interface (you can send any key:value pairs as a POST request).

We do not recommend using this unless you want more advanced functionality.

Keyword Arguments

- `payload`: key-value pairs that will be converted into a POST request.

refresh_login()

Logs in to the PACER system using the login and password provided at the initialization of

search_agent(). This will create a Requests session that will allow you to query the PACER system. If *auto_login* =False, refresh_login() must be called before you can query the case_locator. This function will raise an error if you supply an invalid login or password.

Returns nothing.

request_docket_sheet(*docket_link*, *other_options={}*)
Returns the HTML of the docket sheet specified by *docket_link*.

You can also pass additional POST requests through *other_options*.

request_document(*case_filename*, *document_link*, *other_options={}*)
Using a case_filename and a link to the document, this function constructs the necesssary POST data and finds the correct document URL to download the specified PDF document.

Returns binary data.

You can also pass additional POST requests through *other_options*.

(For version 2.1) Currently only implemented for district courts, but should eventually be implemented for bankruptcy and appellate courts.

search_case_locator(*case_no*, *other_options={'default_form': 'b', 'court_type': 'all'}*)
Passes a query to the PACER Case Locator and returns a list of search results (as well as error message, if applicable). Returns two objects, a list (*results*) and a string that indicates if there was an error.

Keyword Arguments

- case_no: a string that represents a PACER query.
- other_options: allows you to determine the payload sent to query_case_locator(). This is validated in search_case_locator() so that you only pass known valid POST requests. The default options are those known to be necessary to get search results.

Output Documentation Each search result is a dictionary with these keys:

- searched_case_no
- result_no
- case_name
- listed_case_no
- court_id
- nos
- date_filed
- date_closed
- query_link

The second object returned is a string that verbosely indicates errors that occured. If the search result was found, the string is empty.

2.4.2 Other Functions

pacer_lib.scraper.**disaggregate_docket_number**(*combined_docket_number*)
Returns a string that indicates the year of the case and the PACER-valid case_id.

Disaggregates the year from the case number when we have combined docket numbers. Combined year and case numbers are often stored as integers, but this leads to the truncation of leading zeroes. We restore these leading zeroes and then return the two-digit year of the case and the case_id. The minimum number of digits for this

function is five (which assumes that the case was from 2000). If there are further truncations (e.g., '00-00084' stored as '0000084' and truncated to '84'), pre-process your case-numbers.

pacer_lib.scraper.**gen_case_query**(*district*, *office*, *year*, *docket_number*, *type_code*, *district_first=True*)

Creates a PACER query from the district, office, year, case_id and case_type and returns a tuple of (case_id, court_id, region).

PACER case-numbers can be generated by consolidating the district, office, year, case id and case type information in a specific way. This function formats the district name and type_code correctly and then combines the case identifying information into a single PACER query.

Many other data sources list the district of the court before the state, e.g., EDNY rather than NYED. If this is not the case, turn off the district_first option.

Keyword Arguments

- `year` should be either 2 digits (e.g., 00) or 4 digits (e.g., 1978).
- `case_id` should be exactly 5digits
- `type code` must be one of the following: civil, civ, criminal, crim, bankruptcy, bank, cv, cr, bk

Returns a tuple (case_number, court_id)

(For Version 2.1) Note: Appellate Courts have not been implemented yet.

Some of this functionality may not be necessary and should be revisited.

Specifically, year can be 2 or 4 digits and case number does not have to be exactly 5 digits (up to 5 digits). Office must be exactly 1 digit.

We could also consider including the specific sate in the output. We should also create a list of all valid courtids and check against it.

2.5 pacer_lib.reader

class pacer_lib.reader.**UTF8Recoder**(*f*, *encoding*)

Iterator that reads an encoded stream and reencodes the input to UTF-8

class pacer_lib.reader.**UnicodeReader**(*f*, *dialect=<class csv.excel>*, *encoding='utf-8'*, ***kwds*)

A CSV reader which will iterate over lines in the CSV file "f", which is encoded in the given encoding.

class pacer_lib.reader.**UnicodeWriter**(*f*, *dialect=<class csv.excel>*, *encoding='utf-8'*, ***kwds*)

A CSV writer which will write rows to CSV file "f", which is encoded in the given encoding.

class pacer_lib.reader.**docket_parser**(*docket_path='./results/local_docket_archive'*, *output_path='./results'*)

Returns a docket_parser object that provides functions which allow you to quickly load .html PACER docket sheets from the specified docket_path parse metadata (about both the download of the docket as well as the characteristics of the case), and convert into a machine-readable format (CSV)

This object is built on top of BeautifulSoup 4.

Keyword Arguments:

- `docket_path`: which specifies a relative path to the storage of dockets (i.e., input data); dockets shoudl be in .html format
- `output_path`: which specifies a relative path to the folder where output should be written. If this folder does not exist, it will be created. If the two subfolders (`/case_meta/` and `/download_meta`) do not exist within the output_path, then they will also be created.

extract_all_meta(*data*, *debug=False*)

Returns two dictionaries, one that has download_meta and one that contains meta extracted from the docket. `extract_all_meta()` runs `extract_case_meta()`, `extract_lawyer_meta()` and `extract_download_meta()` on `data` (a string literal of an .html document). It returns two dictionaries (one containing download_meta and one containing both case_meta and lawyer_meta) because download_meta and case_meta have overlapping information.

If debug is not turned on, extract_all_meta will ignore any error output from the sub functions (e.g., if the functions cannot find the relevant sections).

Output Documentation See the output documentation of `extract_case_meta()`, `extract_lawyer_meta()` and `extract_download_meta()`.

extract_case_meta(*data*)

Returns a dictionary of case information (e.g., case_name, demand, nature of suit, jurisdiction, assigned judge, etc.) extracted from an .html docket (passed as a string literal through `data`). This information should be available in all dockets downloaded from PACER.

This information may overlap with information from `extract_download_meta()`, but it is technically extracted from a different source (the docket sheet, rather than the results page of the PACER Case Locator).

In consolidated cases, there is information about the lead case, and a link. We extract any links in the case_meta section of the document and store it in the dictionary with the key *meta_links*.

There are some encoding issues with characters such as Ã that we have tried to address, but may need to be improved in the future.

If `extract_case_meta()` cannot find the case_meta section of the docket, it will return a dictionary with a single key, *Error_case_meta*.

Output Documentation Please note that `extract_case_meta` does common cleaning and then treats each (text):(text) line as a key:value pair, so this documentation only documents the most common keys that we have observed.

These keys are, generally, self-explanatory and are only listed for convenience.

- `Case name`
- `Assigned to`
- `Referred to`
- `Demand`
- `Case in other court`
- `Cause`
- `Date Filed`
- `Date Terminated`
- `Jury Demand`
- `Nature of Suit`
- `Jurisdiction`

Special keys:

- `Member case`: the existence of this key indicates that this is probably the lead case of a consolidated case.

- `Lead case`: the existence of this key indicates that this is probably a member case of a consolidated case.
- `meta_links`: this will only exists if there are links in the case_meta section of the PACER docket.

`extract_download_meta`(*data*)

Returns a dictionary that contains all of the downloadmeta that was stored by `pacer_lib.scraper()` at the time of download (i.e., the *detailed_info* json object that is commented out at the top of new downloads from PACER). This is meant to help improve reproducibility.

detailed_info is an add-on in later versions of pacer_lib that records case-level data from the search screen (date_closed, link, nature of suit, case-name, etc.) as well as the date and time of download.

In earlier versions of pacer_lib (i.e., released as pacer_scraper_library), this was stored as a list and did not include the date and time of download. `extract_download_meta()` can also handle these *detailed_info* objects.

If there is no *detailed_info*, the function returns a dictionary with the key 'Error_download_meta'.

Keyword Arguments

- `data`: should be a string, read from a .html file.

Output Documentation Unless otherwise noted, all of these are collected from the PACER Case Locator results page. This is documented as `key`: description of value.

These terms are found in documents downloaded by any version of pacer_lib:

- `searched_case_no`: the case number that was passed to pacer_lib.scraper(), this is recorded to ensure reproducibility and comes from pacer_lib. This is not found on the PACER Case Locator results page.
- `court_id`: the abbreviation for the court the case was located in
- `case_name`: the name of the case, as recorded by PACER
- `nos`: a code for "Nature of Suit"
- `date_filed`: the date the case was filed, as recorded by PACER
- `date_closed`: the date the case was closed, as recorded by PACER
- `link`: a link to the docket

These are only in documents downloaded with newer versions of pacer_lib:

- `downloaded`: string that describes the time the docket was downloaded by pacer_lib. This is not found on the PACER Case Locator results page. (Format: yyyy-mm-dd,hh:mm:ss)
- `listed_case_no`: string that describes the preferred PACER case no for this case (as opposed to the query we submitted)
- `result_no`: which result was the case on the PACER Case Locator results page.

`extract_lawyer_meta`(*data*)

Returns a dictionary of information about the plaintiff, defendant and their lawyers extracted from an .html docket (passed as a string literal through `data`).

At the moment, `extract_lawyer_meta()` only handles the most common listing (i.e., if there is one listing for plaintiff and one listing for defendant). If there is more than one set of plaintiffs or defendants (e.g., in a class action suit), the function will return a dictionary with a single key *Error_lawyer_meta*. This function will not handle movants and will probably not handle class-action cases.

In dockets downloaded from older versions of pacer_lib (e.g., pacer_scraper_library), lawyer information was not requested so the dockets will not contain any lawyer_meta to be extracted.

Output Documentation This is documented as `key`: description of value.

- `plaintiffs`: list of the names of plaintiffs
- `defendants`: list of the names of defendants
- `plaintiffs_attorneys`: list of the name of attorneys representing the plaintiffs
- `defendants_attorneys`: list of the name of attorneys representing the defendants
- `plaintiffs_attorneys_details`: string that contains the cleaned output of all plaintiff lawyer data (e.g., firm, address, email, etc.) that can be further cleaned in the future.
- `defendants_attorneys_details`: string that contains the cleaned output of all defendant lawyer data (e.g., firm, address, email, etc.) that can be further cleaned in the future.

parse_data(*data*)

Returns a list of all of the docket entries in `data`, which should be a string literal. BeautifulSoup is useed to parse a .html docket file (pass as a string literal through `data`) into a list of docket entries. Each docket entry is also a list.

This uses html.parser and, in the case of failure, switches to html5lib.

If it cannot find the table or entries, it will return a string as an error message.

Keyword Arguments

- `data`: should be a string, read from a .html file.

Output Documentation

0.date_filed

1.document_number

2.docket_description

3.link_exist (this is a dummy to indicate the existence of a link)

4.document_link (docket_number does not uniquely identify the docket entry so we also create a separate unique identifier)

5.unique_id (document_number is not a unique identifier so we create one based on the placement in the .html docket sheet)

parse_dir(*overwrite=True*, *get_meta=True*)

Run `parse_data()` and `extract_all_meta()` on each file in the docket_path folder and writes the output to the output_path.

Output Documentation This function returns nothing.

File documentation The docket entries of each docket are stored as a .csv in a folder 'processed_dockets'. The filename of the csv indicates the source docket and the columns represent (in order):

0.date_filed

1.document_number

2.docket_description

3.link_exist (this is a dummy to indicate the existence of a link)

4.document_link (docket_number does not uniquely identify the docket entry so we also create a separate unique identifier)

5.unique_id (document_number is not a unique identifier so we create one based on the placement in the .html docket sheet)

The download meta and case and lawyer meta information of each docket is stored as a JSON-object in the sub-folders 'processed_dockets_meta/download_meta/' and 'processed_dockets_meta/case_meta/' within the output path. The files indicate the source docket and are prefixed by **download_meta_** and **case_meta_**, respectively.

class pacer_lib.reader.**docket_processor**(*processed_path='./results/parsed_dockets'*, *output_path='./results/'*)

Returns a docket_processor() object that allows for keyword and boolean searching of docket entries from dockets specified in *processed_path*. docket_processor relies on the use of *docket_parser'* to parse .html PACER dockets into structured .csv, although it is theoretically possible (but quite tedious) to independently bring dockets into compliance for use with docket_processor.

This will give you a set of documents (and their associated links) for download (and which can be passed to pacer_lib.scraper()).

The object then outputs a docket-level or consolidated .csv that describes all documents that meet the search criteria (stored in *hit_list*).

Keyword Arguments

- processed_path points to the folder containing .csv docket files
- output_path points to the folder where you would like output to be stored. Note that the output will actually be stored in a subfolder of the *output_path* called */docket_hits/*. If the folders do not exist, they will be created.

search_dir(*require_term=[]*, *exclude_term=[]*, *case_sensitive=False*, *within=0*)

Runs search_docket() on each docket in *self.processed_path* and adds hits to *self.hit_list* as a key value pair *case_number* : *[docket entries]*, where *case_number* is taken from the filename and *[docket_entries]* is a list of docket entries (which are also lists) that meet the search criteria.

The search criteria is specified by *require_term*, *exclude_term*, *case_sensitive* and *within*, such that:

- if *within* !=0, all searches are constrained to the first x characters of the text, where x = *within*
- all strings in the list *require_term* are found in *text* (or the first x charactersm, if *within* is used)
- and, no strings in the list *exclude_term* are found in *text* (or the first x charactersm, if *within* is used)
- if *case_sensitive* =True, then the search is case sensitive

Returns nothing.

search_docket(*docket*, *require_term=[]*, *exclude_term=[]*, *case_sensitive=False*, *within=0*)

Returns a lists of docket entries that match the search criteria. Docket entries are lists that should have the same structure as described in docket_parser, i.e. in order:

0.date_filed

1.document_number

2.docket_description

3.link_exist (this is a dummy to indicate the existence of a link)

4.document_link (docket_number does not uniquely identify the docket entry so we also create a separate unique identifier)

5.unique_id (document_number is not a unique identifier so we create one based on the placement in the .html docket sheet)

The docket is specified by the argument *docket* and searched for in the *self.processed_path* folder.

The search criteria is specified by *require_term*, *exclude_term*, *case_sensitive* and *within*, such that:

- if *within* !=0, all searches are constrained to the first x characters of the text, where x = *within*

•all strings in the list *require_term* are found in *text* (or the first x charactersm, if *within* is used)

•and, no strings in the list *exclude_term* are found in *text* (or the first x charactersm, if *within* is used)

•if *case_sensitive* =True, then the search is case sensitive

search_text(*text*, *require_term=[]*, *exclude_term=[]*, *case_sensitive=False*)

Returns a boolean indicating if all criteria are satisified in *text*. The criteria are determined in this way:

•all strings in the list *require_term* are found in *text*

•and, no strings in the list *exclude_term* are found in *text*

If you pass a string instead of a list to either *require_term* or *exclude_term*, `search_text()` will convert it to a list.

This search is, by default case-insensitive, but you can turn on case-sensitive search through *case_sensitive*.

write_all_matches(*suffix*, *overwrite_flag=False*)

Writes all of the matches found in the *self.hit_list* dictionary to a single .csv file (**all_match__[suffix].csv**) in the *self.output_path*. The columns of the .csv are (in order):

0.case_number (as defined by the source .csv)

1.date_filed

2.document_number

3.docket_description

4.link_exist (this is a dummy to indicate the existence of a link)

5.document_link (docket_number does not uniquely identify the docket entry so we also create a separate unique identifier)

6.unique_id (document_number is not a unique identifier so we create one based on the placement in the .html docket sheet)

There is a flag for overwriting.

You cannot use `/ \ % * : | " < > . _` in the suffix.

Returns nothing.

write_individual_matches(*suffix*, *overwrite_flag=False*)

Writes all of the matches in the *self.hit_list* dictionary to one .csv file per docket sheet (determined by the source .csv) in a folder named after the suffix. To distinguish from the source .csv, they are prefixed by a ^. They are also suffixed to allow for multiple searches of the same source .csv.

Suffix is required and if the same suffix is specified, it will overwrite previous searches if the overwrite flag is turned on. (It will delete all of the old files in the suffix folder.)

You cannot use `/ \ % * : | " < > . _` in the suffix.

Returns nothing.

class pacer_lib.reader.**document_sorter**(*docket_path='./results/local_docket_archive'*, *document_path='./results/local_document_archive'*, *output_path='./results'*, *searchable_criteria='court'*)

Not implemented yet. Sorry.

convert_PDF_to_text(*filename*)

Convert a file to text and save it in the text_output_path

convert_all(*overwrite=False*)

For files in the document path, use convert_PDF_to_text if it has not been converted before. Determine if a file is searchable or not.

count()
Count the file_index

export_file_index()
Save the file_index to a file

flag_searchable()
Flag according to self.flags() Move files to a folder (make this an option)

set_flag()
Add a criteria to the flagging process.

2.5.1 docket_parser

class pacer_lib.reader.**docket_parser**(*docket_path='./results/local_docket_archive'*, *output_path='./results'*)
Returns a docket_parser object that provides functions which allow you to quickly load .html PACER docket sheets from the specified docket_path parse metadata (about both the download of the docket as well as the characteristics of the case), and convert into a machine-readable format (CSV)

This object is built on top of BeautifulSoup 4.

Keyword Arguments:

- `docket_path`: which specifies a relative path to the storage of dockets (i.e., input data); dockets shoudl be in .html format
- `output_path`: which specifies a relative path to the folder where output should be written. If this folder does not exist, it will be created. If the two subfolders (`/case_meta/` and `/download_meta`) do not exist within the output_path, then they will also be created.

extract_all_meta(*data*, *debug=False*)
Returns two dictionaries, one that has download_meta and one that contains meta extracted from the docket. `extract_all_meta()` runs `extract_case_meta()`, `extract_lawyer_meta()` and `extract_download_meta()` on `data` (a string literal of an .html document). It returns two dictionaries (one containing download_meta and one containing both case_meta and lawyer_meta) because download_meta and case_meta have overlapping information.

If debug is not turned on, extract_all_meta will ignore any error output from the sub functions (e.g., if the functions cannot find the relevant sections).

Output Documentation See the output documentation of `extract_case_meta()`, `extract_lawyer_meta()` and `extract_download_meta()`.

extract_case_meta(*data*)
Returns a dictionary of case information (e.g., case_name, demand, nature of suit, jurisdiction, assigned judge, etc.) extracted from an .html docket (passed as a string literal through `data`). This information should be available in all dockets downloaded from PACER.

This information may overlap with information from `extract_download_meta()`, but it is technically extracted from a different source (the docket sheet, rather than the results page of the PACER Case Locator).

In consolidated cases, there is information about the lead case, and a link. We extract any links in the case_meta section of the document and store it in the dictionary with the key *meta_links*.

There are some encoding issues with characters such as Ã that we have tried to address, but may need to be improved in the future.

If `extract_case_meta()` cannot find the case_meta section of the docket, it will return a dictionary with a single key, *Error_case_meta*.

Output Documentation Please note that `extract_case_meta` does common cleaning and then treats each (text):(text) line as a key:value pair, so this documentation only documents the most common keys that we have observed.

These keys are, generally, self-explanatory and are only listed for convenience.

- `Case name`
- `Assigned to`
- `Referred to`
- `Demand`
- `Case in other court`
- `Cause`
- `Date Filed`
- `Date Terminated`
- `Jury Demand`
- `Nature of Suit`
- `Jurisdiction`

Special keys:

- `Member case`: the existence of this key indicates that this is probably the lead case of a consolidated case.
- `Lead case`: the existence of this key indicates that this is probably a member case of a consolidated case.
- `meta_links`: this will only exists if there are links in the case_meta section of the PACER docket.

extract_download_meta(*data*)

Returns a dictionary that contains all of the downloadmeta that was stored by `pacer_lib.scraper()` at the time of download (i.e., the *detailed_info* json object that is commented out at the top of new downloads from PACER). This is meant to help improve reproducibility.

detailed_info is an add-on in later versions of pacer_lib that records case-level data from the search screen (date_closed, link, nature of suit, case-name, etc.) as well as the date and time of download.

In earlier versions of pacer_lib (i.e., released as pacer_scraper_library), this was stored as a list and did not include the date and time of download. `extract_download_meta()` can also handle these *detailed_info* objects.

If there is no *detailed_info*, the function returns a dictionary with the key 'Error_download_meta'.

Keyword Arguments

- `data`: should be a string, read from a .html file.

Output Documentation Unless otherwise noted, all of these are collected from the PACER Case Locator results page. This is documented as `key`: description of value.

These terms are found in documents downloaded by any version of pacer_lib:

- `searched_case_no`: the case number that was passed to pacer_lib.scraper(), this is recorded to ensure reproducibility and comes from pacer_lib. This is not found on the PACER Case Locator results page.
- `court_id`: the abbreviation for the court the case was located in

- `case_name`: the name of the case, as recorded by PACER
- `nos`: a code for "Nature of Suit"
- `date_filed`: the date the case was filed, as recorded by PACER
- `date_closed`: the date the case was closed, as recorded by PACER
- `link`: a link to the docket

These are only in documents downloaded with newer versions of pacer_lib:

- `downloaded`: string that describes the time the docket was downloaded by pacer_lib. This is not found on the PACER Case Locator results page. (Format: yyyy-mm-dd,hh:mm:ss)
- `listed_case_no`: string that describes the preferred PACER case no for this case (as opposed to the query we submitted)
- `result_no`: which result was the case on the PACER Case Locator results page.

`extract_lawyer_meta`(*data*)

Returns a dictionary of information about the plaintiff, defendant and their lawyers extracted from an .html docket (passed as a string literal through `data`).

At the moment, `extract_lawyer_meta()` only handles the most common listing (i.e., if there is one listing for plaintiff and one listing for defendant). If there is more than one set of plaintiffs or defendants (e.g., in a class action suit), the function will return a dictionary with a single key *Error_lawyer_meta*. This function will not handle movants and will probably not handle class-action cases.

In dockets downloaded from older versions of pacer_lib (e.g., pacer_scraper_library), lawyer information was not requested so the dockets will not contain any lawyer_meta to be extracted.

Output Documentation This is documented as `key`: description of value.

- `plaintiffs`: list of the names of plaintiffs
- `defendants`: list of the names of defendants
- `plaintiffs_attorneys`: list of the name of attorneys representing the plaintiffs
- `defendants_attorneys`: list of the name of attorneys representing the defendants
- `plaintiffs_attorneys_details`: string that contains the cleaned output of all plaintiff lawyer data (e.g., firm, address, email, etc.) that can be further cleaned in the future.
- `defendants_attorneys_details`: string that contains the cleaned output of all defendant lawyer data (e.g., firm, address, email, etc.) that can be further cleaned in the future.

`parse_data`(*data*)

Returns a list of all of the docket entries in `data`, which should be a string literal. BeautifulSoup is useed to parse a .html docket file (pass as a string literal through `data`) into a list of docket entries. Each docket entry is also a list.

This uses html.parser and, in the case of failure, switches to html5lib.

If it cannot find the table or entries, it will return a string as an error message.

Keyword Arguments

- `data`: should be a string, read from a .html file.

Output Documentation

0.date_filed

1.document_number

2.docket_description

3.link_exist (this is a dummy to indicate the existence of a link)

4.document_link (docket_number does not uniquely identify the docket entry so we also create a separate unique identifier)

5.unique_id (document_number is not a unique identifier so we create one based on the placement in the .html docket sheet)

parse_dir(*overwrite=True*, *get_meta=True*)

Run `parse_data()` and `extract_all_meta()` on each file in the docket_path folder and writes the output to the output_path.

Output Documentation This function returns nothing.

File documentation The docket entries of each docket are stored as a .csv in a folder 'processed_dockets'. The filename of the csv indicates the source docket and the columns represent (in order):

0.date_filed

1.document_number

2.docket_description

3.link_exist (this is a dummy to indicate the existence of a link)

4.document_link (docket_number does not uniquely identify the docket entry so we also create a separate unique identifier)

5.unique_id (document_number is not a unique identifier so we create one based on the placement in the .html docket sheet)

The download meta and case and lawyer meta information of each docket is stored as a JSON-object in the sub-folders 'processed_dockets_meta/download_meta/' and 'processed_dockets_meta/case_meta/' within the output path. The files indicate the source docket and are prefixed by **download_meta_** and **case_meta_**, respectively.

2.5.2 docket_processor

class `pacer_lib.reader.`**docket_processor**(*processed_path='./results/parsed_dockets'*, *output_path='./results/'*)

Returns a `docket_processor()` object that allows for keyword and boolean searching of docket entries from dockets specified in *processed_path*. `docket_processor` relies on the use of *docket_parser*' to parse .html PACER dockets into structured .csv, although it is theoretically possible (but quite tedious) to independently bring dockets into compliance for use with `docket_processor`.

This will give you a set of documents (and their associated links) for download (and which can be passed to pacer_lib.scraper()).

The object then outputs a docket-level or consolidated .csv that describes all documents that meet the search criteria (stored in *hit_list*).

Keyword Arguments

- `processed_path` points to the folder containing .csv docket files
- `output_path` points to the folder where you would like output to be stored. Note that the output will actually be stored in a subfolder of the *output_path* called */docket_hits/*. If the folders do not exist, they will be created.

search_dir(*require_term=[]*, *exclude_term=[]*, *case_sensitive=False*, *within=0*)
Runs `search_docket()` on each docket in *self.processed_path* and adds hits to *self.hit_list* as a key value pair *case_number* : *[docket entries]*, where *case_number* is taken from the filename and *[docket_entries]* is a list of docket entries (which are also lists) that meet the search criteria.

The search criteria is specified by *require_term*, *exclude_term*, *case_sensitive* and *within*, such that:

- if *within* !=0, all searches are constrained to the first x characters of the text, where x = *within*
- all strings in the list *require_term* are found in *text* (or the first x charactersm, if *within* is used)
- and, no strings in the list *exclude_term* are found in *text* (or the first x charactersm, if *within* is used)
- if *case_sensitive* =True, then the search is case sensitive

Returns nothing.

search_docket(*docket*, *require_term=[]*, *exclude_term=[]*, *case_sensitive=False*, *within=0*)
Returns a lists of docket entries that match the search criteria. Docket entries are lists that should have the same structure as described in docket_parser, i.e. in order:

0. date_filed
1. document_number
2. docket_description
3. link_exist (this is a dummy to indicate the existence of a link)
4. document_link (docket_number does not uniquely identify the docket entry so we also create a separate unique identifier)
5. unique_id (document_number is not a unique identifier so we create one based on the placement in the .html docket sheet)

The docket is specified by the argument *docket* and searched for in the *self.processed_path* folder.

The search criteria is specified by *require_term*, *exclude_term*, *case_sensitive* and *within*, such that:

- if *within* !=0, all searches are constrained to the first x characters of the text, where x = *within*
- all strings in the list *require_term* are found in *text* (or the first x charactersm, if *within* is used)
- and, no strings in the list *exclude_term* are found in *text* (or the first x charactersm, if *within* is used)
- if *case_sensitive* =True, then the search is case sensitive

search_text(*text*, *require_term=[]*, *exclude_term=[]*, *case_sensitive=False*)
Returns a boolean indicating if all criteria are satisified in *text*. The criteria are determined in this way:

- all strings in the list *require_term* are found in *text*
- and, no strings in the list *exclude_term* are found in *text*

If you pass a string instead of a list to either *require_term* or *exclude_term*, `search_text()` will convert it to a list.

This search is, by default case-insensitive, but you can turn on case-sensitive search through *case_sensitive*.

write_all_matches(*suffix*, *overwrite_flag=False*)
Writes all of the matches found in the *self.hit_list* dictionary to a single .csv file (**all_match__[suffix].csv**) in the *self.output_path*. The columns of the .csv are (in order):

0. case_number (as defined by the source .csv)
1. date_filed
2. document_number

3.docket_description

4.link_exist (this is a dummy to indicate the existence of a link)

5.document_link (docket_number does not uniquely identify the docket entry so we also create a separate unique identifier)

6.unique_id (document_number is not a unique identifier so we create one based on the placement in the .html docket sheet)

There is a flag for overwriting.

You cannot use / \ % * : | " < > . _ in the suffix.

Returns nothing.

write_individual_matches(*suffix*, *overwrite_flag=False*)

Writes all of the matches in the *self.hit_list* dictionary to one .csv file per docket sheet (determined by the source .csv) in a folder named after the suffix. To distinguish from the source .csv, they are prefixed by a ^. They are also suffixed to allow for multiple searches of the same source .csv.

Suffix is required and if the same suffix is specified, it will overwrite previous searches if the overwrite flag is turned on. (It will delete all of the old files in the suffix folder.)

You cannot use / \ % * : | " < > . _ in the suffix.

Returns nothing.

2.5.3 document_sorter

class pacer_lib.reader.**document_sorter**(*docket_path='./results/local_docket_archive'*, *document_path='./results/local_document_archive'*, *output_path='./results'*, *searchable_criteria='court'*)

Not implemented yet. Sorry.

convert_PDF_to_text(*filename*)

Convert a file to text and save it in the text_output_path

convert_all(*overwrite=False*)

For files in the document path, use convert_PDF_to_text if it has not been converted before. Determine if a file is searchable or not.

count()

Count the file_index

export_file_index()

Save the file_index to a file

flag_searchable()

Flag according to self.flags() Move files to a folder (make this an option)

set_flag()

Add a criteria to the flagging process.

2.6 FAQ

No one has asked any questions yet. If you want to ask a question, go ahead and contact us on our Github page.

2.7 Changelog

Version 2.33 (2014-03-19)

Added truncation so that overlong docket descriptions do not break `docket_parser()`

Changed the default folder for parsed docket sheets from `\processed_dockets\` to `\parsed_dockets\` to eliminate confusion

Added csv headers to output files.

Began address bug #7

Version 2.32, 2.31 (2014-02-26)

Fixed three bugs in `reader.docket_parser().parse_data`, `reader.docket_parser.extract_download_meta()` and `reader.docket_parser.extract_case_meta(` by implementing html5lib as an alternative processor and adding some string handling for quotation marks.

Fixed bug in `reader.docket_processor.search_text()` that would convert strings into single-item lists.

Version 2.3, 2.2, 2.1 (2014-02-18)

Made a bunch of mistakes, fixed them (mostly of the packaging variety) but burned through Versions 2.1 and 2.2.

Changed the name of submodule pacer_lib.parser to pacer_lib.reader because of potential confusion.

Implemented overwrite protection and suffixing for `docket_processor.write_all_matches()`

Implemented overwrite protection for `docket_processor.write_individual_matches()`

Cleaned up the documentation.

Version 2.0 (2014-02-17)

Added `parser` sub-module, which includes the objects `docket_parser()` and `docket_processor()`, which brings scraping and docket parsing functionality to

`document_sorter()` outlined in `parser` sub-module but not yet implemented.

Improved documentation, including the use of docstrings, Sphinx and hosting documentation on ReadTheDocs.org.

Kevin Jiang added as maintainer.

Version 1.0 (2014-01-08)

Added `scraper` sub-module, which includes the object `search_agent()` that interfaces with the PACER Case Locator and allows the downloading of both dockets and documents.

Added the functions `disaggregate_docket_number()` and `gen_case_query`, which handle specific query-creation issues in our PACER requests.

***pacer_scraper_library* Version 1.0a (2013-03-01)** Original library; function based. For legacy users, you can access the undocumented and unsupported pacer_scraper_library here.

- genindex

p

C

D

E

F

G

P

Q

R

S

U

W