medicare$_{u}tilsDocumentation$

Release 0.0.1

Kyle Barron

Nov 29, 2018

Contents:

A Python package to make working with Medicare data easier.

CHAPTER 1

Installation

This package only supports Python 3.6 or higher. You can find out the version of Python installed by running `python --version` in your terminal. The first two numbers must be 3.6 or 3.7.

```
$ python --version
Python 3.6.4 :: Anaconda custom (64-bit)
```

1.1 Stable release

To install medicare_utils, run this command in your terminal:

```
$ pip install medicare_utils --upgrade
```

This is the preferred method to install medicare_utils, as it will always install the most recent stable release.

If you don't have `pip` installed, I recommend installing the Anaconda distribution, which will install a wide variety of helpful data science packages. Otherwise, this Python installation guide can guide you through the process of installing `pip` manually.

1.2 Development version

If you want the newest version available, you can install direct from the Github repository with:

```
$ pip install git+https://github.com/kylebarron/medicare_utils --upgrade
```

1.3 From sources

The sources for medicare_utils can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/kylebarron/medicare_utils
```

Or download the tarball:

```
$ curl  -OL https://github.com/kylebarron/medicare_utils/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 2

Quick Start guide

2.1 Importing the package

First, make sure you've installed `medicare_utils`. Then to use the package, you need to import it:

```
import medicare_utils as med
```

The `as med` means that you can refer to the package as `med` instead of `medicare_utils` from here on.

2.2 Data extracts

Data extracts are started with the `med.MedicareDF` function. For example, I can begin an extract using 1% sample data and for the years 2010-2012 with:

```
mdf = med.MedicareDF(percent=1, years=range(2010, 2013))
```

Note that the `range` function includes integers up to but not including the second argument.

Then I can get a cohort of white women aged 66-75

In recent years, Python has become the `fastest growing major programming language <https://stackoverflow.blog/2017/09/06/incredible-growth-python/>`*, largely due to its widespread use among data scientists. This popularity has fostered packages that work with data, such as* `Pandas <https://pandas.pydata.org/>`, the standard for in-memory data analysis. A newer package, `Dask <https://dask.pydata.org/en/latest/dataframe.html>`_, has been developed to parallelize Pandas operations and work with data larger than memory.

CHAPTER 3

Usage

To use `medicare_utils` in a project:

```
import medicare_utils as med
```

CHAPTER 4

API

4.1 medicare_utils.codebook

`medicare_utils.codebook.`**`codebook`**`(`*data_type*`)`

Load variable codebook

Parameters **`data_type`** (`str`) – Type of file to get codebook for

- `bsfab` (Beneficiary Summary File, Base segment)
- `med` (MedPAR File)
- `opc` (Outpatient File, Claims segment)

Return type `dict`

Returns `dict` with variable names as keys; values are another `dict`. This inner `dict` has two keys: `name`, where the value is the descriptive name of the variable, and `values`, which is itself a `dict` with variable values as keys and value descriptions as values of the `dict`.

Examples

To get the labels of the values of `clm_type`, in the `med` file, you could do

```
>>> import medicare_utils as med
>>> cbk = med.codebook('med')['clm_type']['values']
```

Now `cbk` is a `dict` where the keys are the values the variable can take, and the values are the labels of the variable's values.

```
>>> from pprint import pprint
>>> pprint(cbk)
{'10': 'HHA claim',
 '20': 'Non swing bed SNF claim',
 '30': 'Swing bed SNF claim',
```

(continues on next page)

(continued from previous page)

```
'40': 'Outpatient claim',
'50': 'Hospice claim',
'60': 'Inpatient claim',
'61': "Inpatient 'Full-Encounter' claim",
'62': 'Medicare Advantage IME/GME claims',
'63': 'Medicare Advantage (no-pay) claims',
'64': 'Medicare Advantage (paid as FFS) claim',
'71': 'RIC O local carrier non-DMEPOS claim',
'72': 'RIC O local carrier DMEPOS claim',
'81': 'RIC M DMERC non-DMEPOS claim',
'82': 'RIC M DMERC DMEPOS claim'}
```

4.2 medicare_utils.codes

class `medicare_utils.codes.`**hcpcs**(*year*, *path=”*)

Bases: `object`

A class to work with HCPCS codes

class `medicare_utils.codes.`**icd9**(*year*, *long=True*, *path=”*)

Bases: `object`

A class to work with ICD9 codes

class `medicare_utils.codes.`**npi**(*columns=None*, *regex=None*, *download=False*, *path=”*, *load=True*)

Bases: `object`

A class to work with NPI codes

load(*columns=None*, *regex=None*)

Return type `DataFrame`

4.3 medicare_utils.medicare_df

class `medicare_utils.medicare_df.`**MedicareDF**(*percent*, *years*, *year_type=’calendar’*, *dask=False*, *verbose=False*, *parquet_engine=’pyarrow’*, *max_cores=None*, *pq_path=’/disk/agebulk3/medicare.work/doyle-DUA51929/barronk-DUA51929/raw/pq’*)

Bases: `object`

Create a MedicareDF object.

Parameters

- **percent** (`Union[str, int, float]`) – Percent sample of data to use. As a numeric value, either `0.01`, `1`, `5`, `20`, or `100`. If a string, must be either `'0001'`, `'01'`, `'05'`, `'20'`, or `'100'`.
- **years** (`Union[int, List[int]]`) – Years of data to use.

  If `year_type` is `'calendar'`, all data within calendar years provided will be used for the *`get_cohort()`* and *`search_for_codes()`* methods.

If `year_type` is `'age'`, each year of exported data starts on the patient's birthday of that year. For example if `years` is `[2008, 2009, 2010]` and `year_type` is `'age'`, the exported data with label `2008` will be derived starting from each patient's birthday in 2008 up through the day before the patient's birthday in 2009. Because this discards the data before a patient's birthday in the first year and after a patient's birthday in the last year, providing `n` years will result in `n-1` years in the output. Patients who were born on February 29th in a leap year include data from that date through February 28th of the following year.

- **year_type** (`str`) – `calendar` to work with multiple years as calendar years; `age` to work with patients' age years. The latter does computations using the age a patient is when admitted.
- **dask** (`bool`) – Use dask library for out of core computation. In general this is unnecessary because this package tries to be smart about only storing the minimum amount of data in memory at a time. This is a global option that applies for all methods called from this object. However, as of now, dask support doesn't exist for *`search_for_codes()`*.
- **verbose** (`bool`) – Print progress status of program to console. This is a global option that applies for all methods called from this object.
- **parquet_engine** (`str`) – The engine to use to read Parquet files: `pyarrow` or `fastparquet`. Usually you'll have better reliability when using the engine you created the Parquet files with.
- **max_cores** (`Optional[int]`) – Maximum number of CPU cores to use. By default uses `multiprocessing.cpu_count() - 1`.
- **pq_path** (`str`) – Path to Parquet Medicare files. As of now, these must be created by hand using *`medicare_utils.parquet.convert_med()`*, but I hope to have these available for NBER use in the future.

Returns `MedicareDF` object. Can then create a cohort based on demographic characteristics using *`get_cohort()`* or search for claims with given diagnosis or procedure codes with *`search_for_codes()`*.

Examples

Create a `MedicareDF` object by passing the desired arguments to it as a function, and assigning the result to a new variable.

```
>>> import medicare_utils as med
>>> mdf = med.MedicareDF(percent=5, years=[2010, 2011, 2012], year_type='calendar
↪', verbose=True)
```

or

```
>>> import medicare_utils as med
>>> mdf = med.MedicareDF(percent=100, years=range(2010, 2013), year_type='age',␣
↪verbose=True)
```

Note: in Python, `range` doesn't include the upper bound, so `range(2010, 2013)` is equivalent to `[2010, 2011, 2012]`.

The `mdf` object now lets you find a demographic cohort with `mdf.get_cohort()` or search for claims with given diagnosis or procedure codes with *`search_for_codes()`*.

`get_cohort`(*gender=None, ages=None, races=None, rti_race=False, buyin_val=None, hmo_val=None, join='outer', keep_vars=[], dask=False, verbose=False*)
Get cohort with given demographic and enrollment characteristics.

Creates `.pl` attribute with patient-level data in the form of a pandas DataFrame. Unless

Parameters

- **gender** (`Optional[str]`) – Gender of patients to keep. Options: `'M'`, `'F'`, `'Male'`, `'Female'`, or `None` (keep both).
- **ages** (`Union[int, List[int], None]`) – Ages of patients to keep. When `year_type` is `'calendar'`, keeps anyone whose age was in `ages` at the end of the calendar year. When `year_type` is `'age'`, keeps anyone whose age was in `ages` at any point during the year.
- **races** (`Union[List[str], str, None]`) – Races to keep. Strings such as `'white'`, `'black'`, `'asian'`, and `'hispanic'` are allowed. Alternatively, can be the integers those strings correspond to in the data codebook. The codebook is here for the regular race code or here for the RTI race code.
- **rti_race** (`bool`) – If `True`, uses the Research Triangle Institute race code instead of the default race code.
- **buyin_val** (`Union[List[str], str, None]`) – The values `buyinXX` can take. When `year_type` is `'calendar'`, keeps everyone whose buyin value is in `buyin_val` for the entire calendar year. When `year_type` is `'age'`, keeps everyone whose buyin value is in `buyin_val` for the 13 months beginning in the month of the patient's birthday. See the codebook here.
- **hmo_val** (`Union[List[str], str, None]`) – The values `hmoindXX` can take. When `year_type` is `'calendar'`, keeps everyone whose HMO indicator is in `hmo_val` for the entire calendar year. When `year_type` is `'age'`, keeps everyone whose HMO indicator is in `hmo_val` for the 13 months beginning in the month of the patient's birthday. See the codebook here.
- **join** (`str`) – Method for joining across years. Meaningless when `years` is a single year.
  - `'outer'` keeps all patients who matched desired characteristics in **any** year.
  - `'inner'` keeps all patients who matched desired characteristics in **all** years.
  - `'left'` keeps all patients who matched desired characteristics in the **first** year.
  - `'right'` keeps all patients who matched desired characteristics in the **last** year.
- **keep_vars** (`Union[str, Pattern[AnyStr], List[Union[str, Pattern[AnyStr]]], None]`) – Variable names to keep in final output. This can either be a string or a list of strings or compiled regular expressions. The easiest way to create a compiled regular expression is with `re.compile('string').` By default, the only columns returned from `get_cohort` are the patient identifier (either `bene_id` or `ehic`) and a variable named `match_{year}` that is `True` if the patient was found in a given year and `False` otherwise. The list of variables in the dataset can be found here.
- **dask** (`bool`) – Use dask library for out of core computation. In general this is unnecessary because this package tries to be smart about only storing the minimum amount of data in memory at a time.
- **verbose** (`bool`) – Print progress of program to console.

Returns Creates attributes `.pl` with patient-level data in pandas DataFrame and `.nobs_dropped` with dict of percent of observations dropped due to each filter. Index of DataFrame is always `bene_id`, even if years provided are before 2006. In pre-2006 years, `ehic` will always be returned as a column.

Examples

First, set up the class by running *MedicareDF*.

```
>>> import medicare_utils as med
>>> mdf = med.MedicareDF(percent=100, years=range(2008, 2012))
```

Then use the mdf object to supply your parameters. To find all female patients who are Asian or White, and who are aged 70-85 (inclusive) and continuously enrolled in Medicare Part A and B in any of the years 2008-2011 (inclusive), we can do:

```
>>> mdf.get_cohort(
        gender='female',
        ages=range(70, 86),
        races=['asian', 'white'],
        buyin_val=['3', 'C'],
        join='outer')
```

In case we wanted to add any extra columns from the Beneficiary summary files to this extract, we could pass variable names as a list with the keep_vars argument. By default, the only columns returned from get_cohort are the patient identifier and an indicator for if the patient was found in a given year.

```
>>> mdf.get_cohort(
        gender='female',
        ages=range(70, 86),
        races=['asian', 'white'],
        buyin_val=['3', 'C'],
        join='outer',
        keep_vars=['age', 'bene_dob', 'state_cd', 'zip_cd'])
```

To keep all the buyin* variables in the extract, we can additionally pass a compiled regular expression as an argument:

```
>>> import re
>>> mdf.get_cohort(
        gender='female',
        ages=range(70, 86),
        races=['asian', 'white'],
        buyin_val=['3', 'C'],
        join='outer',
        keep_vars=[
            'age', 'bene_dob', 'state_cd', 'zip_cd',
            re.compile(r'^buyin\d\d')])
```

Then the data is held within the pl attribute, and can be accessed like any other object.

```
>>> len(mdf.pl)
```

search_for_codes(*data_types*, *pl=None*, *hcpcs=None*, *icd9_dx=None*, *icd9_dx_max_cols=None*, *icd9_sg=None*, *icd9_sg_max_cols=None*, *keep_vars={}*, *collapse_codes=True*, *rename={'hcpcs': None, 'icd9_dx': None, 'icd9_sg': None}*, *convert_ehic=True*, *dask=False*, *verbose=False*)

Search in claim-level datasets for HCPCS and/or ICD9 codes

Note: Each code given must be distinct, or collapse_codes must be True.

Parameters

- **data_types** (Union[str, List[str]]) – Files to search through. The following are allowed:
  - carc (Carrier File, Claims segment)
  - carl (Carrier File, Line segment)
  - ipc (Inpatient File, Claims segment)
  - ipr (Inpatient File, Revenue Center segment)
  - med (MedPAR File)
  - opc (Outpatient File, Claims segment)
  - opr (Outpatient File, Revenue Center segment)
- **pl** (Union[DataFrame, Index, None]) – Patient-level DataFrame used to filter cohort before searching code columns. Unnecessary if *get_cohort()* is called before this. Must have at least ehic or bene_id as a column.
- **hcpcs** (Union[str, Pattern[AnyStr], List[Union[str, Pattern[AnyStr]]], None]) – HCPCS codes to search for
- **icd9_dx** (Union[str, Pattern[AnyStr], List[Union[str, Pattern[AnyStr]]], None]) – ICD-9 diagnosis codes to search for
- **icd9_dx_max_cols** (Optional[int]) – Max number of ICD9 diagnosis code columns to search through. If None, will search through all columns.
- **icd9_sg** (Union[str, Pattern[AnyStr], List[Union[str, Pattern[AnyStr]]], None]) – ICD-9 procedure codes to search for
- **icd9_sg_max_cols** (Optional[int]) – Max number of ICD9 procedure code columns to search through. If None, will search through all columns.
- **keep_vars** (Dict[str, Union[str, Pattern[AnyStr], List[Union[str, Pattern[AnyStr]]], None]]) – Variable names to keep in final output. This must be a dictionary where the keys are one of the data_types being searched in and the values of the dictionary are strings or lists of strings or compiled regular expressions. The easiest way to create a compiled regular expression is with re.compile(string).

  By default, the only columns returned from search_for_codes are 1) a column named match that is True if any of the codes were matched for a given claim and False otherwise, and 2) columns for each code or regular expression given if collapse_codes is False. The lists of variables in each dataset can be found at the links under the data_types argument.
- **collapse_codes** (bool) – If True, the returned DataFrame will have a single column named match that is True for the claims where any supplied code was matched and False otherwise. If collapse_codes is False, the returned DataFrame will contain a column for each string or regular expression provided. Use the rename argument to give user friendly names to columns with possibly complex regular expressions.
- **rename** (Dict[str, Union[str, None, List[str], Dict[str, str]]]) – Match columns to rename when collapse_codes is False.
- **convert_ehic** (bool) – If True, merges on bene_id for years 2005 and earlier. If False, will leave ehic as the patient identifier. This is always True when years before and after 2005 are provided.
- **dask** (bool) – Use dask library for out of core computation. Not yet implemented; as of now everything happens in core.

- **verbose** (bool) – Print progress of program to console

Returns Creates .cl attribute. This is a dict where keys are the data_types provided and values are pandas DataFrames with bene_id as index and indicator columns for each code provided.

Examples

First, set up the class by running *MedicareDF* and optionally using *get_cohort()*.

```
>>> import medicare_utils as med
>>> mdf = med.MedicareDF(percent=100, years=range(2008, 2012))
>>> mdf.get_cohort(gender='female', ages=range(70, 86))
```

Then to perform a search for claims:

- in the MedPAR or Outpatient claims files
- with the primary or secondary ICD-9 diagnosis code starting with either 410 or 480
- returning individual match columns for each of the two provided regular expressions and renaming them to ami and pneumonia

we can do:

```
>>> import re
>>> mdf.search_for_codes(
        data_types=['med', 'opc'],
        icd9_dx=[re.compile(r'^410'), re.compile(r'^480')],
        icd9_dx_max_cols=2,
        collapse_codes=False,
        rename={'icd9_dx': ['ami', 'pneumonia']})
```

In case we wanted to add any extra columns from the Beneficiary summary files to this extract, we could pass variable names to the keep_vars argument. Since the function needs to know which dataset to get the columns from, this argument must be a dictionary, not a list. To keep the admsndt and dschrgdt variables from the MedPAR file and the from_dt and thru_dt variables from the Outpatient claims file, we can do:

```
>>> import re
>>> mdf.search_for_codes(
        data_types=['med', 'opc'],
        icd9_dx=[re.compile(r'^410'), re.compile(r'^480')],
        icd9_dx_max_cols=2,
        collapse_codes=False,
        rename={'icd9_dx': ['ami', 'pneumonia']},
        keep_vars={
            'med': ['admsndt', 'dschrgdt'],
            'opc': ['from_dt', 'thru_dt']})
```

To include all the diagnosis code columns from the MedPAR file in the extract, we can additionally pass a compiled regular expression to keep_vars that selects all variables of the format dgnscd##:

```
>>> import re
>>> mdf.search_for_codes(
        data_types=['med', 'opc'],
        icd9_dx=[re.compile(r'^410'), re.compile(r'^480')],
```

(continues on next page)

(continued from previous page)

```
          icd9_dx_max_cols=2,
          collapse_codes=False,
          rename={'icd9_dx': ['ami', 'pneumonia']},
          keep_vars={
               'med': ['admsndt', 'dschrgdt',
                       re.compile(r'^dgnscd(\d+)$')],
               'opc': ['from_dt', 'thru_dt']})
```

Then the data is held as a dictionary within the `cl` attribute, where the keys of the dictionary are the `data_types` provided to the function, and the values of the dictionary are the DataFrames.

```
>>> len(mdf.cl['med'])
>>> len(mdf.cl['opc'])
```

to_stata(*attr*, ***kwargs*)

Wrapper to export to stata.

Parameters

- **attr** (`str`) – str either 'pl' or 'cl.med', 'cl.opc', 'cl.opr', etc.
- **fname** – path (string), buffer or path object string, path object (pathlib.Path or py._path.local.LocalPath) or object implementing a binary write() functions. If using a buffer then the buffer will not be automatically closed after the file data has been written.
- **convert_dates** – dict Dictionary mapping columns containing datetime types to stata internal format to use when writing the dates. Options are 'tc', 'td', 'tm', 'tw', 'th', 'tq', 'ty'. Column can be either an integer or a name. Datetime columns that do not have a conversion type specified will be converted to 'tc'. Raises NotImplementedError if a datetime column has timezone information.
- **encoding** – str Default is latin-1. Unicode is not supported.
- **time_stamp** – datetime A datetime to use as file creation date. Default is the current time.
- **data_label** – str A label for the data set. Must be 80 characters or smaller.
- **variable_labels** – dict Dictionary containing columns as keys and variable labels as values. Each label must be 80 characters or smaller.
- **version** – {114, 117} Version to use in the output dta file. Version 114 can be used read by Stata 10 and later. Version 117 can be read by Stata 13 or later. Version 114 limits string variables to 244 characters or fewer while 117 allows strings with lengths up to 2,000,000 characters.
- **convert_strl** – list, optional List of column names to convert to string columns to Stata StrL format. Only available if version is 117. Storing strings in the StrL format can produce smaller dta files if strings have more than 8 characters and values are repeated.

Examples

```
>>> mdf.to_stata(attr='pl', fname='patient_level_file.dta')
>>> mdf.to_stata(attr='cl.med', fname='medpar_extract.dta')
```

Or with dates

```
>>> mdf.to_stata(attr='cl.med', fname='medpar_extract.dta', convert_dates={
↪'admsndt': 'td'})
```

4.4 medicare_utils.parquet

`medicare_utils.parquet.`**`convert_file`**(*infile*, *outfile*, *rename_dict=None*, *rg_size=2.5*, *parquet_engine='pyarrow'*, *compression_type='SNAPPY'*, *manual_schema=False*, *ehic_xw=None*)

Convert arbitrary Stata file to Parquet format

Parameters

- **`infile`** (`str`) – path of file to read from
- **`outfile`** (`str`) – path of file to export to
- **`rename_dict`** (`Optional[Dict[str, str]]`) – keys should be initial variable names; values should be new variable names
- **`rg_size`** (`float`) – Size in GB of the individual row groups
- **`parquet_engine`** (`str`) – either `pyarrow` or `fastparquet`
- **`compression_type`** (`str`) – Compression algorithm to use. Can be `SNAPPY` or `GZIP`.
- **`manual_schema`** (`bool`) – Create parquet schema manually. For use with pyarrow; doesn't always work
- **`ehic_xw`** (`Optional[str]`) – Merge bene_id onto old files with ehic

Return type `None`

Returns Writes .parquet file to disk.

`medicare_utils.parquet.`**`convert_med`**(*pcts=['0001', '01', '05', '100'], years=range(2001, 2013), data_types=['carc', 'opc', 'bsfab', 'med'], rg_size=2.5, parquet_engine='pyarrow', compression_type='SNAPPY', manual_schema=False, ehic_xw=True, n_jobs=6, med_dta='/disk/aging/medicare/data', med_pq='/disk/agebulk3/medicare.work/doyle-DUA51929/barronk-DUA51929/raw/pq'*)

Convert Medicare Stata files to parquet

Parameters

- **`pcts`** (`Union[str, List[str]]`) – percent samples to convert
- **`years`** (`Union[int, List[int]]`) – file years to convert
- **`data_types`** (`Union[str, List[str]]`) – types of data files to convert
  - `bsfab` (Beneficiary Summary File, Base segment)
  - `bsfcc` (Beneficiary Summary File, Chronic Conditions segment)
  - `bsfcu` (Beneficiary Summary File, Cost & Use segment)
  - `bsfd` (Beneficiary Summary File, National Death Index segment)
  - `carc` (Carrier File, Claims segment)
  - `carl` (Carrier File, Line segment)

  - den (Denominator File)
  - dmec (Durable Medical Equipment File, Claims segment)
  - dmel (Durable Medical Equipment File, Line segment)
  - hhac (Home Health Agency File, Claims segment)
  - hhar (Home Health Agency File, Revenue Center segment)
  - hosc (Hospice File, Claims segment)
  - hosr (Hospice File, Revenue Center segment)
  - ipc (Inpatient File, Claims segment)
  - ipr (Inpatient File, Revenue Center segment)
  - med (MedPAR File)
  - opc (Outpatient File, Claims segment)
  - opr (Outpatient File, Revenue Center segment)
  - snfc (Skilled Nursing Facility File, Claims segment)
  - snfr (Skilled Nursing Facility File, Revenue Center segment)
  - xw (Crosswalks files for `ehic` - `bene_id`)
- **rg_size** (`float`) – size in GB of each Parquet row group
- **parquet_engine** (`str`) – either 'fastparquet' or 'pyarrow'
- **compression_type** (`str`) – 'SNAPPY' or 'GZIP'
- **manual_schema** (`bool`) – whether to create manual parquet schema. Doesn't always work.
- **ehic_xw** (`bool`) – Merge bene_id onto old files with ehic
- **n_jobs** (`int`) – number of processes to use
- **med_dta** (`str`) – top of tree for medicare stata files
- **med_pq** (`str`) – top of tree to output new parquet files

Return type `None`

4.5 medicare_utils.utils

`medicare_utils.utils.`**`fpath`**(*percent*, *year*, *data_type*, *dta=False*, *dta_path='/disk/aging/medicare/data'*, *pq_path='/disk/agebulk3/medicare.work/doyle-DUA51929/barronk-DUA51929/raw/pq'*)

Generate path to Medicare files

Parameters

- **percent** (`Union[float, int, str]`) – percent sample of data
- **year** (`Union[int, str]`) – year of data
- **data_type** (`str`) – desired type of file
  - bsfab (Beneficiary Summary File, Base segment)

    - `bsfcc` (Beneficiary Summary File, Chronic Conditions segment)
    - `bsfcu` (Beneficiary Summary File, Cost & Use segment)
    - `bsfd` (Beneficiary Summary File, National Death Index segment)
    - `carc` (Carrier File, Claims segment)
    - `carl` (Carrier File, Line segment)
    - `den` (Denominator File)
    - `dmec` (Durable Medical Equipment File, Claims segment)
    - `dmel` (Durable Medical Equipment File, Line segment)
    - `hhac` (Home Health Agency File, Claims segment)
    - `hhar` (Home Health Agency File, Revenue Center segment)
    - `hosc` (Hospice File, Claims segment)
    - `hosr` (Hospice File, Revenue Center segment)
    - `ipc` (Inpatient File, Claims segment)
    - `ipr` (Inpatient File, Revenue Center segment)
    - `med` (MedPAR File)
    - `opc` (Outpatient File, Claims segment)
    - `opr` (Outpatient File, Revenue Center segment)
    - `snfc` (Skilled Nursing Facility File, Claims segment)
    - `snfr` (Skilled Nursing Facility File, Revenue Center segment)
    - `xw` (Crosswalks files for `ehic` - `bene_id`)
    - `xw_bsf` (Crosswalks files for `ehic` - `bene_id`)
- **`dta`** (`bool`) – Returns Stata file path
- **`dta_path`** (`str`) – top of tree for medicare stata files
- **`pq_path`** (`str`) – top of tree for medicare parquet files

Return type `str`

Returns path to file

`medicare_utils.utils.`**`pq_vars`**`(`*ParquetFile*`)`

CHAPTER 5

Credits

5.1 Development Lead

- Kyle Barron barronk@mit.edu

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

Contributing

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

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/kylebarron/medicare_utils/issues.

If you are reporting a bug, please include:

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

6.1.2 Fix Bugs or Implement Features

Look through the GitHub issues for bugs. Most issues are open to whoever wants to implement it, but comment on it so that I know you're working on it.

6.1.3 Write Documentation

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

6.1.4 Submit Feedback

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

If you are proposing a feature:

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

6.2 Get Started!

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

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

```
$ git clone git@github.com:your_github_user/medicare_utils.git
```

3. Install your local copy into a Conda environment. If you don't have Conda installed, install Anaconda or Miniconda first. Then set up your fork for local development with:

```
$ cd medicare_utils/
$ conda create env -f environment.yml
$ source activate medicare_utils
$ python setup.py develop
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

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

```
$ flake8 medicare_utils tests
$ python setup.py test or py.test
$ tox
```

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

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

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

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

1. The pull request should include tests.

2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.6 and 3.7. Check https://travis-ci.org/kylebarron/medicare_utils/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_medicare_utils
```

6.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in `CHANGELOG.md`). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 7

Changelog

7.1 0.0.1 (2018-03-01)

- First release on PyPI.

CHAPTER 8

Introduction

Medicare data are large and unwieldy. Since the size of the data is often larger than memory, many people use SAS. However SAS is an ugly language and not enjoyable to work with. Python is an easier and faster alternative.

Creating a data extract can be done in three lines of code:

```python
import re
import medicare_utils as med
mdf = med.MedicareDF(
    percent='100',
    years=range(2008, 2014))
mdf.get_cohort(
    gender='male',
    ages=range(65, 75),
    buyin_val=['3', 'C'],
    join='outer',
    keep_vars=['bene_dob'])
mdf.search_for_codes(
    data_types=['med', 'opc'],
    icd9_dx=re.compile(r'^410'),
    icd9_dx_max_cols=1,
    collapse_codes=True,
    keep_vars={'med': ['medparid', 'admsndt', 'dschrgdt']},
    rename={'icd9_dx': 'ami'})
```

The resulting data extract consists of a patient-level file of patients who are:

- Male
- Aged 65-74 (inclusive) in any year from 2008 to 2013
- Continuously enrolled in fee-for-service Medicare in any year from 2008 to 2013 (i.e. `buyin_val` either `3` or `C`)

and a claim-level file of patients who were included in the above cohort and furthermore had a primary diagnosis code of AMI in either the MedPAR or Outpatient claims files. The patient-level file is accessed with `mdf.pl` and the claim-level file is accessed with `mdf.cl`.

This package also provides:

- Classes to work with NPI, ICD-9, and HCPCS codes. These commands will automatically download these data files for you.[1]
- Codebooks for values of categorical variables.
- A simple interface to convert data files from Stata format to the modern Parquet format.

This documentation aims to walk through everything needed to run these routines. Then you can keep working with these extracts in Python or easily export them to Stata's `.dta` format. Head to the *Quick Start guide* to get started.

[1] Datasets with HCPCS codes and short descriptions from 2003 to the present are freely available on the CMS website in their Relative Value Files. These CMS files are released under the End User Point and Click Agreement. In order to not run afoul of this license agreement, this package does not distribute HCPCS codes. Rather, it provides code for the user to download and work with them. By using the HCPCS functions in this package, you agree to the above Agreement.

CHAPTER 9

Caveats

This package contains no Medicare data or private information. It assumes you already have access to Medicare data.

This package was originally developed for use on the National Bureau of Economic Research's servers, but portions of the package may be useful for third parties as well.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index