Exploratory data analysis with pandas in python, part 1

Andras Zsom, Lead Data Scientist at CCV

github.com/brown-ccv/dscov_data_science

Series of talks by yours truly

- DATA1030 - Hands-on Data Science
  - how to develop ML pipelines for tabular data from scratch
- walk through a modified version of the course material
- today:
  - how to manipulate tabular data with pandas
- down the road:
  - how to visualize your data
  - ML overview
  - discuss each step of an ML pipeline

Data transformations: pandas data frames

By the end of this presentation, you will be able to

- read in csv, excel, and sql data into a pandas data frame
- filter rows in various ways
- select columns
- merge and append data frames

Some notes and advice

- **ALWAYS READ THE HELP OF THE METHODS/FUNCTIONS YOU USE!**
- stackoverflow is your friend, use it! https://stackoverflow.com/ (https://stackoverflow.com/)

Pandas

- data are often distributed over multiple files/databases (e.g., csv and excel files, sql databases)
- each file/database is read into a pandas dataframe
- you often need to filter dataframes (select specific rows/columns based on index or condition)
- pandas dataframes can be merged and appended

Data transformations: pandas data frames

By the end of this talk, you will be able to

- **read in csv, excel, and sql data into a pandas data frame**
- filter rows in various ways
- select columns
- merge and append data frames

In [1]:

```
# how to read in a database into a dataframe and basic dataframe structure
import pandas as pd

# load data from a csv file
df = pd.read_csv('data/adult_data.csv') # there are also pd.read_excel(), and pd.read_sql()

#print(df)
#print(df.head()) # by default, shows the first five rows but check help(df.head) to specify the number of rows to show
#print(df.shape) # the shape of your dataframe (number of rows, number of columns)
#print(df.shape[0]) # number of rows
print(df.shape[1]) # number of columns
```

```
15
```

Packages

A package is a collection of classes and functions.

- a dataframe (pd.DataFrame()) is a pandas class
  - a class is the blueprint of how the data should be organized
  - classes have methods which can perform operations on the data (e.g., .head(), .shape)
- df is an object, an instance of the class.
  - we put data into the class
  - methods are attached to objects
    - you cannot call pd.head(), you can only call df.head()
- read_csv is a function
  - functions are called from the package
  - you cannot call df.read_csv, you can only call pd.read_csv()

DataFrame structure: both rows and columns are indexed!

- index column, no name
  - contains the row names
  - by default, index is a range object from 0 to number of rows - 1
  - any column can be turned into an index, so indices can be non-number, and also non-unique. more on this later.
- columns with column names on top

Always print your dataframe to check if it looks ok!

Most common reasons it might not look ok:

- the first row is not the column name
  - there are rows above the column names that need to be skipped
  - there is no column name but by default, pandas assumes the first row is the column name. as a result, the values of the first row end up as column names.
- character encoding is off
- separator is not comma but some other charachter

In [2]:

```
# check the help to find the solution
help(pd.read_csv)
```

```
Help on function read_csv in module pandas.io.parsers:

read_csv(filepath_or_buffer:Union[str, pathlib.Path, IO[~AnyStr]], sep
=',', delimiter=None, header='infer', names=None, index_col=None, useco
ls=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None,
engine=None, converters=None, true_values=None, false_values=None, skip
initialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=
None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_l
ines=True, parse_dates=False, infer_datetime_format=False, keep_date_co
l=False, date_parser=None, dayfirst=False, cache_dates=True, iterator=F
alse, chunksize=None, compression='infer', thousands=None, decimal=
b'.', lineterminator=None, quotechar='"', quoting=0, doublequote=True,
escapechar=None, comment=None, encoding=None, dialect=None, error_bad_l
ines=True, warn_bad_lines=True, delim_whitespace=False, low_memory=Tru
e, memory_map=False, float_precision=None)
    Read a comma-separated values (csv) file into DataFrame.

    Also supports optionally iterating or breaking of the file
    into chunks.

    Additional help can be found in the online docs for
    `IO Tools <http://pandas.pydata.org/pandas-docs/stable/user_guide/i
o.html>`_.

    Parameters
    ----------
    filepath_or_buffer : str, path object or file-like object
        Any valid string path is acceptable. The string could be a URL.
Valid
        URL schemes include http, ftp, s3, and file. For file URLs, a h
ost is
        expected. A local file could be: file://localhost/path/to/tabl
e.csv.

        If you want to pass in a path object, pandas accepts any ``os.P
athLike``.

        By file-like object, we refer to objects with a ``read()`` meth
od, such as
        a file handler (e.g. via builtin ``open`` function) or ``String
IO``.
    sep : str, default ','
        Delimiter to use. If sep is None, the C engine cannot automatic
ally detect
        the separator, but the Python parsing engine can, meaning the l
atter will
        be used and automatically detect the separator by Python's buil
tin sniffer
        tool, ``csv.Sniffer``. In addition, separators longer than 1 ch
aracter and
        different from ``'\s+'`` will be interpreted as regular express
ions and
        will also force the use of the Python parsing engine. Note that
regex
        delimiters are prone to ignoring quoted data. Regex example: `
`'\r\t'``.
    delimiter : str, default ``None``
```

```
        Alias for sep.
    header : int, list of int, default 'infer'
        Row number(s) to use as the column names, and the start of the
        data.  Default behavior is to infer the column names: if no nam
es
        are passed the behavior is identical to ``header=0`` and column
        names are inferred from the first line of the file, if column
        names are passed explicitly then the behavior is identical to
        ``header=None``. Explicitly pass ``header=0`` to be able to
        replace existing names. The header can be a list of integers th
at
        specify row locations for a multi-index on the columns
        e.g. [0,1,3]. Intervening rows that are not specified will be
        skipped (e.g. 2 in this example is skipped). Note that this
        parameter ignores commented lines and empty lines if
        ``skip_blank_lines=True``, so ``header=0`` denotes the first li
ne of
        data rather than the first line of the file.
    names : array-like, optional
        List of column names to use. If file contains no header row, th
en you
        should explicitly pass ``header=None``. Duplicates in this list
are not
        allowed.
    index_col : int, str, sequence of int / str, or False, default ``No
ne``
      Column(s) to use as the row labels of the ``DataFrame``, either g
iven as
      string name or column index. If a sequence of int / str is given,
a
      MultiIndex is used.

      Note: ``index_col=False`` can be used to force pandas to *not* us
e the first
      column as the index, e.g. when you have a malformed file with del
imiters at
      the end of each line.
    usecols : list-like or callable, optional
        Return a subset of the columns. If list-like, all elements must
either
        be positional (i.e. integer indices into the document columns)
or strings
        that correspond to column names provided either by the user in
`names` or
        inferred from the document header row(s). For example, a valid
list-like
        `usecols` parameter would be ``[0, 1, 2]`` or ``['foo', 'bar',
'baz']``.
        Element order is ignored, so ``usecols=[0, 1]`` is the same as
``[1, 0]``.
        To instantiate a DataFrame from ``data`` with element order pre
served use
        ``pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]`` f
or columns
        in ``['foo', 'bar']`` order or
        ``pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']]``
        for ``['bar', 'foo']`` order.
        If callable, the callable function will be evaluated against th
e column
        names, returning names where the callable function evaluates to
True. An
        example of a valid callable argument would be ``lambda x: x.upp
er() in
        ['AAA', 'BBB', 'DDD']``. Using this parameter results in much f
aster
        parsing time and lower memory usage.
    squeeze : bool, default False
        If the parsed data only contains one column then return a Serie
s.
    prefix : str, optional
        Prefix to add to column numbers when no header, e.g. 'X' for X
0, X1, ...
    mangle_dupe_cols : bool, default True
        Duplicate columns will be specified as 'X', 'X.1', ...'X.N', ra
ther than
        'X'...'X'. Passing in False will cause data to be overwritten i
f there
        are duplicate names in the columns.
    dtype : Type name or dict of column -> type, optional
        Data type for data or columns. E.g. {'a': np.float64, 'b': np.i
nt32,
        'c': 'Int64'}
        Use `str` or `object` together with suitable `na_values` settin
gs
        to preserve and not interpret dtype.
        If converters are specified, they will be applied INSTEAD
        of dtype conversion.
    engine : {'c', 'python'}, optional
        Parser engine to use. The C engine is faster while the python e
ngine is
        currently more feature-complete.
    converters : dict, optional
        Dict of functions for converting values in certain columns. Key
s can either
        be integers or column labels.
    true_values : list, optional
        Values to consider as True.
    false_values : list, optional
        Values to consider as False.
    skipinitialspace : bool, default False
        Skip spaces after delimiter.
    skiprows : list-like, int or callable, optional
        Line numbers to skip (0-indexed) or number of lines to skip (in
t)
        at the start of the file.

        If callable, the callable function will be evaluated against th
e row
        indices, returning True if the row should be skipped and False
otherwise.
        An example of a valid callable argument would be ``lambda x: x
in [0, 2]``.
    skipfooter : int, default 0
```

```
        Number of lines at bottom of file to skip (Unsupported with eng
ine='c').
    nrows : int, optional
        Number of rows of file to read. Useful for reading pieces of la
rge files.
    na_values : scalar, str, list-like, or dict, optional
        Additional strings to recognize as NA/NaN. If dict passed, spec
ific
        per-column NA values.  By default the following values are inte
rpreted as
        NaN: '', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-Na
N', '-nan',
        '1.#IND', '1.#QNAN', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan',
        'null'.
    keep_default_na : bool, default True
        Whether or not to include the default NaN values when parsing t
he data.
        Depending on whether `na_values` is passed in, the behavior is
as follows:

        * If `keep_default_na` is True, and `na_values` are specified,
`na_values`
          is appended to the default NaN values used for parsing.
        * If `keep_default_na` is True, and `na_values` are not specifi
ed, only
          the default NaN values are used for parsing.
        * If `keep_default_na` is False, and `na_values` are specified,
only
          the NaN values specified `na_values` are used for parsing.
        * If `keep_default_na` is False, and `na_values` are not specif
ied, no
          strings will be parsed as NaN.

        Note that if `na_filter` is passed in as False, the `keep_defau
lt_na` and
        `na_values` parameters will be ignored.
    na_filter : bool, default True
        Detect missing value markers (empty strings and the value of na
_values). In
        data without any NAs, passing na_filter=False can improve the p
erformance
        of reading a large file.
    verbose : bool, default False
        Indicate number of NA values placed in non-numeric columns.
    skip_blank_lines : bool, default True
        If True, skip over blank lines rather than interpreting as NaN
values.
    parse_dates : bool or list of int or names or list of lists or dic
t, default False
        The behavior is as follows:

        * boolean. If True -> try parsing the index.
        * list of int or names. e.g. If [1, 2, 3] -> try parsing column
s 1, 2, 3
          each as a separate date column.
        * list of lists. e.g.  If [[1, 3]] -> combine columns 1 and 3 a
nd parse as
```

```
          a single date column.
        * dict, e.g. {'foo' : [1, 3]} -> parse columns 1, 3 as date and
call
          result 'foo'

        If a column or index cannot be represented as an array of datet
imes,
        say because of an unparseable value or a mixture of timezones,
the column
        or index will be returned unaltered as an object data type. For
        non-standard datetime parsing, use ``pd.to_datetime`` after
        ``pd.read_csv``. To parse an index or column with a mixture of
timezones,
        specify ``date_parser`` to be a partially-applied
        :func:`pandas.to_datetime` with ``utc=True``. See
        :ref:`io.csv.mixed_timezones` for more.

        Note: A fast-path exists for iso8601-formatted dates.
    infer_datetime_format : bool, default False
        If True and `parse_dates` is enabled, pandas will attempt to in
fer the
        format of the datetime strings in the columns, and if it can be
inferred,
        switch to a faster method of parsing them. In some cases this c
an increase
        the parsing speed by 5-10x.
    keep_date_col : bool, default False
        If True and `parse_dates` specifies combining multiple columns
then
        keep the original columns.
    date_parser : function, optional
        Function to use for converting a sequence of string columns to
an array of
        datetime instances. The default uses ``dateutil.parser.parser``
to do the
        conversion. Pandas will try to call `date_parser` in three diff
erent ways,
        advancing to the next if an exception occurs: 1) Pass one or mo
re arrays
        (as defined by `parse_dates`) as arguments; 2) concatenate (row
-wise) the
        string values from the columns defined by `parse_dates` into a
single array
        and pass that; and 3) call `date_parser` once for each row usin
g one or
        more strings (corresponding to the columns defined by `parse_da
tes`) as
        arguments.
    dayfirst : bool, default False
        DD/MM format dates, international and European format.
    cache_dates : boolean, default True
        If True, use a cache of unique, converted dates to apply the da
tetime
        conversion. May produce significant speed-up when parsing dupli
cate
        date strings, especially ones with timezone offsets.
        .. versionadded:: 0.25.0
    iterator : bool, default False
        Return TextFileReader object for iteration or getting chunks wi
th
        ``get_chunk()``.
    chunksize : int, optional
        Return TextFileReader object for iteration.
        See the `IO Tools docs
        <http://pandas.pydata.org/pandas-docs/stable/io.html#io-chunkin
g>`_
        for more information on ``iterator`` and ``chunksize``.
    compression : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default
'infer'
        For on-the-fly decompression of on-disk data. If 'infer' and
        `filepath_or_buffer` is path-like, then detect compression from
the
        following extensions: '.gz', '.bz2', '.zip', or '.xz' (otherwis
e no
        decompression). If using 'zip', the ZIP file must contain only
one data
        file to be read in. Set to None for no decompression.

        .. versionadded:: 0.18.1 support for 'zip' and 'xz' compressio
n.

    thousands : str, optional
        Thousands separator.
    decimal : str, default '.'
        Character to recognize as decimal point (e.g. use ',' for Europ
ean data).
    lineterminator : str (length 1), optional
        Character to break file into lines. Only valid with C parser.
    quotechar : str (length 1), optional
        The character used to denote the start and end of a quoted ite
m. Quoted
        items can include the delimiter and it will be ignored.
    quoting : int or csv.QUOTE_* instance, default 0
        Control field quoting behavior per ``csv.QUOTE_*`` constants. U
se one of
        QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE
_NONE (3).
    doublequote : bool, default ``True``
       When quotechar is specified and quoting is not ``QUOTE_NONE``, i
ndicate
       whether or not to interpret two consecutive quotechar elements I
NSIDE a
       field as a single ``quotechar`` element.
    escapechar : str (length 1), optional
        One-character string used to escape other characters.
    comment : str, optional
        Indicates remainder of line should not be parsed. If found at t
he beginning
        of a line, the line will be ignored altogether. This parameter
must be a
        single character. Like empty lines (as long as ``skip_blank_lin
es=True``),
        fully commented lines are ignored by the parameter `header` but
```

```
not by
        `skiprows`. For example, if ``comment='#'``, parsing
        ``#empty\na,b,c\n1,2,3`` with ``header=0`` will result in 'a,b,
c' being
        treated as the header.
    encoding : str, optional
        Encoding to use for UTF when reading/writing (ex. 'utf-8'). `Li
st of Python
        standard encodings
        <https://docs.python.org/3/library/codecs.html#standard-encodin
gs>`_ .
    dialect : str or csv.Dialect, optional
        If provided, this parameter will override values (default or no
t) for the
        following parameters: `delimiter`, `doublequote`, `escapechar`,
        `skipinitialspace`, `quotechar`, and `quoting`. If it is necess
ary to
        override values, a ParserWarning will be issued. See csv.Dialec
t
        documentation for more details.
    error_bad_lines : bool, default True
        Lines with too many fields (e.g. a csv line with too many comma
s) will by
        default cause an exception to be raised, and no DataFrame will
be returned.
        If False, then these "bad lines" will dropped from the DataFram
e that is
        returned.
    warn_bad_lines : bool, default True
        If error_bad_lines is False, and warn_bad_lines is True, a warn
ing for each
        "bad line" will be output.
    delim_whitespace : bool, default False
        Specifies whether or not whitespace (e.g. ``' '`` or ``'    '`
`) will be
        used as the sep. Equivalent to setting ``sep='\s+'``. If this o
ption
        is set to True, nothing should be passed in for the ``delimiter
``
        parameter.

        .. versionadded:: 0.18.1 support for the Python parser.

    low_memory : bool, default True
        Internally process the file in chunks, resulting in lower memor
y use
        while parsing, but possibly mixed type inference.  To ensure no
mixed
        types either set False, or specify the type with the `dtype` pa
rameter.
        Note that the entire file is read into a single DataFrame regar
dless,
        use the `chunksize` or `iterator` parameter to return the data
in chunks.
        (Only valid with C parser).
    memory_map : bool, default False
        If a filepath is provided for `filepath_or_buffer`, map the fil
```

```
e object
        directly onto memory and access the data directly from there. U
sing this
        option can improve performance because there is no longer any
I/O overhead.
    float_precision : str, optional
        Specifies which converter the C engine should use for floating-
point
        values. The options are `None` for the ordinary converter,
        `high` for the high-precision converter, and `round_trip` for t
he
        round-trip converter.

    Returns
    -------
    DataFrame or TextParser
        A comma-separated values (csv) file is returned as two-dimensio
nal
        data structure with labeled axes.

    See Also
    --------
    to_csv : Write DataFrame to a comma-separated values (csv) file.
    read_csv : Read a comma-separated values (csv) file into DataFrame.
    read_fwf : Read a table of fixed-width formatted lines into DataFra
me.

    Examples
    --------
    >>> pd.read_csv('data.csv')  # doctest: +SKIP
```

Exercise 1

How should we read in adult_test.csv properly? Identify and fix the problem.

In [3]:

```
df = pd.read_csv('data/adult_test.csv')

print(df.head())
```

```
  This is the test set for the adult dataset.  Unnamed: 1 Unnamed: 2  \
0     The first two lines need to be skipped.         NaN        NaN   
1                                         age   workclass     fnlwgt   
2                                          25     Private     226802   
3                                          38     Private      89814   
4                                          28   Local-gov     336951   

    Unnamed: 3     Unnamed: 4           Unnamed: 5           Unnamed: 6  
\
0          NaN            NaN                  NaN                  NaN   
1    education  education-num       marital-status           occupation   
2         11th              7        Never-married    Machine-op-inspct   
3      HS-grad              9   Married-civ-spouse      Farming-fishing   
4   Assoc-acdm             12   Married-civ-spouse      Protective-serv   

     Unnamed: 7 Unnamed: 8 Unnamed: 9   Unnamed: 10  Unnamed: 11  \
0           NaN        NaN        NaN           NaN          NaN   
1  relationship       race        sex  capital-gain capital-loss   
2     Own-child      Black       Male             0            0   
3       Husband      White       Male             0            0   
4       Husband      White       Male             0            0   

      Unnamed: 12     Unnamed: 13   Unnamed: 14  
0             NaN             NaN           NaN  
1  hours-per-week  native-country  gross-income  
2              40   United-States        <=50K.  
3              50   United-States        <=50K.  
4              40   United-States         >50K.  
```

In [4]:
```python
# two solutions
df = pd.read_csv('data/adult_test.csv',header=2)
df = pd.read_csv('data/adult_test.csv',skiprows=2)
print(df.head())
```

```
   age   workclass  fnlwgt      education  education-num        marital-
status  \
0   25     Private  226802           11th              7         Never-m
arried
1   38     Private   89814        HS-grad              9   Married-civ-
spouse
2   28   Local-gov  336951     Assoc-acdm             12   Married-civ-
spouse
3   44     Private  160323   Some-college             10   Married-civ-
spouse
4   18           ?  103497   Some-college             10         Never-m
arried

           occupation relationship    race      sex  capital-gain   \
0   Machine-op-inspct    Own-child   Black     Male             0
1     Farming-fishing      Husband   White     Male             0
2     Protective-serv      Husband   White     Male             0
3   Machine-op-inspct      Husband   Black     Male          7688
4                   ?    Own-child   White   Female             0

   capital-loss  hours-per-week  native-country gross-income
0             0              40   United-States       <=50K.
1             0              50   United-States       <=50K.
2             0              40   United-States        >50K.
3             0              40   United-States        >50K.
4             0              30   United-States       <=50K.
```

Data transformations: pandas data frames

By the end of this talk, you will be able to

- read in csv, excel, and sql data into a pandas data frame
- **filter rows in various ways**
- select columns
- merge and append data frames

How to select rows?

1) Integer-based indexing, numpy arrays are indexed the same way.

2) Select rows based on the value of the index column

3) select rows based on column condition

1) Integer-based indexing, numpy arrays are indexed the same way.

In [5]:

```
# df.iloc[] - for more info, see https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#indexing-integer
# iloc is how numpy arrays are indexed (non-standard python indexing)

# [start:stop:step] -  general indexing format

# start stop step are optional
#print(df.iloc[:])
#print(df.iloc[::])
#print(df.iloc[::1])

# select one row - 0-based indexing
#print(df.iloc[3])

# indexing from the end of the data frame
#print(df.iloc[-2])
```

In [6]:

```
# select a slice - stop index not included
#print(df.iloc[3:7])

# select every second element of the slice - stop index not included
# print(df.iloc[3:7:2])

#print(df.iloc[3:7:-2]) # return empty dataframe
#print(df.iloc[7:3:-2])#  return rows with indices 7 and 5. 3 is the stop so it is not included

# can be used to reverse rows
#print(df.iloc[::-1])

# here is where indexing gets non-standard python
# select the 2nd, 5th, and 10th rows
#print(df.iloc[[1,4,9]]) # such indexing doesn't work with lists but it works with numpy arrays
```

2) Select rows based on the value of the index column

In [7]:
```
# df.loc[] - for more info, see https://pandas.pydata.org/pandas-docs/st
able/user_guide/indexing.html#indexing-label

#print(df.index) # the default index when reading in a file is a range i
ndex. In this case,
                 # .loc and .iloc works ALMOST the same.
# one difference:
#print(df.loc[3:9:2]) # this selects the 4th, 6th, 8th, 10th rows - the
 stop element is included!

#help(df.set_index)
```

In [8]:
```
# df_index_age = df.set_index('age',drop=False)

#print(df_index_age.index)
#print(df_index_age.head())

# print(df_index_age.loc[30].head()) # collect everyone with age 30 - th
e index is non-unique

# print(df_index_age.loc[30:35]) # non-default index cannot be sliced.
                               # this does not return everyone between a
ges of 30 and 35
```

3) select rows based on column condition

In [9]:
```
# one condition
#print(df[df['age']==30].head())
# here is the condition: it's a boolean series - series is basically a d
ataframe with one column
#print(df['age']==30)

# multiple conditions can be combined with & (and) | (or)
#print(df[(df['age']>30)&(df['age']<35)].head())
#print(df[(df['age']==90)|(df['native-country']==' Hungary')])
```

Exercise 2

How many people in adult_data.csv work at least 60 hours a week and have a doctorate?

In [10]:

```
# solution
df = pd.read_csv('data/adult_data.csv')
print(df[(df['hours-per-week'] >= 60)&(df['education']==' Doctorate')].shape[0])
# [96 rows x 15 columns]
# we will learn how to modify columns and remove irregularities like this later.

# mistakes the students could make:
print(df[(df['hours-per-week'] >= 60)&(df['education']=='Doctorate')].shape)

print(df[(df['hours-per-week'] > 60)&(df['education']==' Doctorate')].shape)

df = pd.read_csv('data/adult_test.csv',skiprows=2)
print(df[(df['hours-per-week'] >= 60)&(df['education']==' Doctorate')].shape)

print(df[(df['hours-per-week'] > 60)&(df['education']==' Doctorate')].shape)

# these are all good possibilities for tophat answers
```

```
96
(0, 15)
(39, 15)
(33, 15)
(11, 15)
```

Data transformations: pandas data frames

By the end of this talk, you will be able to

- read in csv, excel, and sql data into a pandas data frame
- filter rows in various ways
- **select columns**
- merge and append data frames

In [11]:

```
columns =  df.columns
print(columns)

# select columns by column name
#print(df[['age','hours-per-week']])
#print(columns[[1,5,7]])
#print(df[columns[[1,5,7]]])

# select columns by index using iloc
#print(df.iloc[:,3])

# select columns by index - not standard python indexing
#print(df.iloc[:,[3,5,6]])

# select columns by index -  standard python indexing
#print(df.iloc[:,::2])
```

```
Index(['age', 'workclass', 'fnlwgt', 'education', 'education-num',
       'marital-status', 'occupation', 'relationship', 'race', 'sex',
       'capital-gain', 'capital-loss', 'hours-per-week', 'native-countr
y',
       'gross-income'],
      dtype='object')
```

Data transformations: pandas data frames

By the end of this talk, you will be able to

- read in csv, excel, and sql data into a pandas data frame
- filter rows in various ways
- select columns
- **merge and append data frames**

How to merge dataframes?

Merge - info on data points are distributed in multiple files

```
In [12]: # We have two datasets from two hospitals

         hospital1 = {'ID':['ID1','ID2','ID3','ID4','ID5','ID6','ID7'],'col1':[5,
         8,2,6,0,2,5],'col2':['y','j','w','b','a','b','t']}
         df1 = pd.DataFrame(data=hospital1)
         print(df1)

         hospital2 = {'ID':['ID2','ID5','ID6','ID10','ID11'],'col3':[12,76,34,98,
         65],'col2':['q','u','e','l','p']}
         df2 = pd.DataFrame(data=hospital2)
         print(df2)
```

```
    ID  col1 col2
0  ID1     5    y
1  ID2     8    j
2  ID3     2    w
3  ID4     6    b
4  ID5     0    a
5  ID6     2    b
6  ID7     5    t
     ID  col3 col2
0   ID2    12    q
1   ID5    76    u
2   ID6    34    e
3  ID10    98    l
4  ID11    65    p
```

```
In [13]: # we are interested in only patients from hospital1
         # df_left = df1.merge(df2,how='left',on='ID') # IDs from the left datafr
         ame (df1) are kept
         # print(df_left)

         # we are interested in only patients from hospital2
         # df_right = df1.merge(df2,how='right',on='ID') # IDs from the right dat
         aframe (df2) are kept
         # print(df_right)

         # we are interested in patiens who were in both hospitals
         # df_inner = df1.merge(df2,how='inner',on='ID') # merging on IDs present
         in both dataframes
         # print(df_inner)

         # we are interested in all patients who visited at least one of the hosp
         itals
         # df_outer = df1.merge(df2,how='outer',on='ID')  # merging on IDs presen
         t in any dataframe
         # print(df_outer)
```

How to append dataframes?

Append - new data comes in over a period of time. E.g., one file per month/quarter/fiscal year etc.

You want to combine these files into one data frame.

In [14]:

```
df_append = df1.append(df2) # note that rows with ID2, ID5, and ID6  are
duplicated! Indices are duplicated too.
print(df_append)

# df_append = df1.append(df2,ignore_index=True) # note that rows with ID
2, ID5, and ID6  are duplicated!
# print(df_append)

# d3 = {'ID':['ID23','ID94','ID56','ID17'],'col1':['rt','h','st','n
e'],'col2':[23,86,23,78]}
# df3 = pd.DataFrame(data=d3)
# print(df3)

# df_append = df1.append([df2,df3],ignore_index=True) # multiple datafra
mes can be appended to df1
# print(df_append)
```

```
     ID  col1 col2  col3
0   ID1   5.0    y   NaN
1   ID2   8.0    j   NaN
2   ID3   2.0    w   NaN
3   ID4   6.0    b   NaN
4   ID5   0.0    a   NaN
5   ID6   2.0    b   NaN
6   ID7   5.0    t   NaN
0   ID2   NaN    q  12.0
1   ID5   NaN    u  76.0
2   ID6   NaN    e  34.0
3  ID10   NaN    l  98.0
4  ID11   NaN    p  65.0

/anaconda3/envs/datasci_v0.0.2_local4/lib/python3.6/site-packages/panda
s/core/frame.py:7138: FutureWarning: Sorting because non-concatenation
axis is not aligned. A future version
of pandas will change to not sort by default.

To accept the future behavior, pass 'sort=False'.

To retain the current behavior and silence the warning, pass 'sort=Tru
e'.

  sort=sort,
```

Exercise 3

In [15]:

```
raw_data_1 = {
        'subject_id': ['1', '2', '3', '4', '5'],
        'first_name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
        'last_name': ['Anderson', 'Ackerman', 'Ali', 'Aoni', 'Atiches']}

raw_data_2 = {
        'subject_id': ['6', '7', '8', '9', '10'],
        'first_name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
        'last_name': ['Bonder', 'Black', 'Balwner', 'Brice', 'Btisan']}

raw_data_3 = {
        'subject_id': ['1', '2', '3', '4', '5', '7', '8', '9', '10', '1
1'],
        'test_id': [51, 15, 15, 61, 16, 14, 15, 1, 61, 16]}

# Create three data frames from raw_data_1, 2, and 3.
# Append the first two data frames and assign it to df_append.
# Merge the third data frame with df_append such that only subject_ids f
rom df_append are present.
# Assign the new data frame to df_merge.
# How many rows and columns do we have in df_merge?
```

In [16]:

```
# The solution

df1 = pd.DataFrame(raw_data_1)
df2 = pd.DataFrame(raw_data_2)
df3 = pd.DataFrame(raw_data_3)

df_append = df1.append(df2)
print(df_append)

df_merge = df_append.merge(df3,how='left',on='subject_id')
print(df_merge)
print(df_merge.shape)
```

```
  subject_id first_name last_name
0          1       Alex  Anderson
1          2        Amy  Ackerman
2          3      Allen       Ali
3          4      Alice      Aoni
4          5     Ayoung   Atiches
0          6      Billy    Bonder
1          7      Brian     Black
2          8       Bran   Balwner
3          9      Bryce     Brice
4         10      Betty    Btisan
  subject_id first_name last_name  test_id
0          1       Alex  Anderson     51.0
1          2        Amy  Ackerman     15.0
2          3      Allen       Ali     15.0
3          4      Alice      Aoni     61.0
4          5     Ayoung   Atiches     16.0
5          6      Billy    Bonder      NaN
6          7      Brian     Black     14.0
7          8       Bran   Balwner     15.0
8          9      Bryce     Brice      1.0
9         10      Betty    Btisan     61.0
(10, 4)
```

Always check that the resulting dataframe is what you wanted to end up with!

- small toy datasets are ideal to test your code.

If you need to do a more complicated dataframe operation, check out pd.concat()!

We will learn how to add/delete/modify columns later when we learn about feature engineering.

By now, you are able to

- read in csv, excel, and sql data into a pandas data frame
- filter rows in various ways
- select columns
- merge and append data frames

```
In [ ]:
```