Mud card

- **I hope we could spend more time in coding examples rather than conceptual review. For example, in today's lecture, when covering steps of ML pipeline, I hope we could have more time go through real examples in the jupyter notebook.**
- **Could we cover the bias-variance tradeoff as it relates to the pipeline we looked at in class with the toys? That was never explicitly mentioned.**
  - I won't always have time to go through the code line by line.
  - The expectation is that you'll study between classes and work with the code yourself.
  - You can also come to the review office hours on Fridays 2-4pm. The TAs will walk through the coding examples and answer all your questions.
- **Generally, it felt like we went through the bias-variance tradeoff example quickly. I didn't totally understand the mechanics of the cross-validation/tuning the "hyperparameters" (what is a hyperparameter?) process, though I felt like I could interpret the plot of the tradeoff.**
  - rewatch the video and work with the coding example after class
  - but we will also revisit this concept many times once we learn about ML models and hyperparameters.
- **I'm a bit confused on why model interpretability helps one to debug one's code**
  - Yeah, I had to rush at the end of the lecture a bit
  - Interpretability helps to understand how your model works, what features contribute the most to making predictions.
  - If you see that the model relies on an unexpected feature that the doman experts can't explain or didn't expect, that's usually a sign for a bug or some other issue in your code.
  - Interpretability also helps to identify racial and gender biases in your model.
- **Generally, I understand the train-validation-test split breakup of the data, but I'd like to get a better understanding of it and feel more comfortable.¬† Particularly with how the validation and test data differ!**
  - We cover data splitting in 2 weeks! :)
- **I felt a little confused as to how the generalization error is used to help us make a better model. My thoughts were that it helps to ensure that the model is consistent throughout all the data, but I'm not sure.**
  - Calculating the generalization error does not make your model better.
  - It tells you how well your model is expected to perform on previously unseen data once it is deployed.
- **When choosing the structured/tabular data for our individual project, what structure of the data would you expect in terms of how many dependant and independent variables there are?**
  - Those are statistician terms :)
  - There needs to be one target variable you want to predict (either regression or classification), and as many features as you want, there is no upper limit on that.
- **Confused on difference between hold out set and test set. Are they the same?**
  - The three sets are either train/validation/test or train/test/holdout. It's a naming difference. The third set is used to calculate the generalization error irrespective of how it is called.

Exploratory data analysis in python, part 1

The steps

1. Exploratory Data Analysis (EDA): you need to understand your data and verify that it doesn't contain errors

- do as much EDA as you can!

2. Split the data into different sets: most often the sets are train, validation, and test (or holdout)

- practitioners often make errors in this step!
- you can split the data randomly, based on groups, based on time, or any other non-standard way if necessary to answer your ML question

3. Preprocess the data: ML models only work if X and Y are numbers! Some ML models additionally require each feature to have 0 mean and 1 standard deviation (standardized features)

- often the original features you get contain strings (for example a gender feature would contain 'male', 'female', 'non-binary', 'unknown') which needs to transformed into numbers
- often the features are not standardized (e.g., age is between 0 and 100) but it needs to be standardized

4. Choose an evaluation metric: depends on the priorities of the stakeholders

- often requires quite a bit of thinking and ethical considerations

5. Choose one or more ML techniques: it is highly recommended that you try multiple models

- start with simple models like linear or logistic regression
- try also more complex models like nearest neighbors, support vector machines, random forest, etc.

6. Tune the hyperparameters of your ML models (aka cross-validation)

- ML techniques have hyperparameters that you need to optimize to achieve best performance
- for each ML model, decide which parameters to tune and what values to try
- loop through each parameter combination
  - train one model for each parameter combination
  - evaluate how well the model performs on the validation set
- take the parameter combo that gives the best validation score
- evaluate that model on the test set to report how well the model is expected to perform on previously unseen data

7. Interpret your model: black boxes are often not useful

- check if your model uses features that make sense (excellent tool for debugging)
- often model predictions are not enough, you need to be able to explain how the model arrived to a particular prediction (e.g., in health care)

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

Some notes and advice

- **ALWAYS READ THE HELP OF THE METHODS/FUNCTIONS YOU USE!**
- stackoverflow is your friend, use it! https://stackoverflow.com/
- you can also use generative AI (like github copilot, bard, or chatGPT's code interpreter) to help you fix bugs

Data transformations: pandas data frames

By the end of this lecture, 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

Data transformations: pandas data frames

By the end of this lecture, 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)
#help(df.head)
#print(df.head(10)) # by default, shows the first five rows but check help(df.head) to specify the number of rows t
#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.readers:

read_csv(filepath_or_buffer: 'FilePath | ReadCsvBuffer[bytes] | ReadCsvBuffer[str]', *, sep: 'str | None | lib.NoDef
ault' = <no_default>, delimiter: 'str | None | lib.NoDefault' = None, header: "int | Sequence[int] | None | Literal
['infer']" = 'infer', names: 'Sequence[Hashable] | None | lib.NoDefault' = <no_default>, index_col: 'IndexLabel | Li
teral[False] | None' = None, usecols: 'UsecolsArgType' = None, dtype: 'DtypeArg | None' = None, engine: 'CSVEngine |
None' = None, converters: 'Mapping[Hashable, Callable] | None' = None, true_values: 'list | None' = None, false_valu
es: 'list | None' = None, skipinitialspace: 'bool' = False, skiprows: 'list[int] | int | Callable[[Hashable], bool]
| None' = None, skipfooter: 'int' = 0, nrows: 'int | None' = None, na_values: 'Hashable | Iterable[Hashable] | Mappi
ng[Hashable, Iterable[Hashable]] | None' = None, keep_default_na: 'bool' = True, na_filter: 'bool' = True, verbose:
'bool | lib.NoDefault' = <no_default>, skip_blank_lines: 'bool' = True, parse_dates: 'bool | Sequence[Hashable] | No
ne' = None, infer_datetime_format: 'bool | lib.NoDefault' = <no_default>, keep_date_col: 'bool | lib.NoDefault' = <n
o_default>, date_parser: 'Callable | lib.NoDefault' = <no_default>, date_format: 'str | dict[Hashable, str] | None'
= None, dayfirst: 'bool' = False, cache_dates: 'bool' = True, iterator: 'bool' = False, chunksize: 'int | None' = No
ne, compression: 'CompressionOptions' = 'infer', thousands: 'str | None' = None, decimal: 'str' = '.', lineterminato
r: 'str | None' = None, quotechar: 'str' = '"', quoting: 'int' = 0, doublequote: 'bool' = True, escapechar: 'str | N
one' = None, comment: 'str | None' = None, encoding: 'str | None' = None, encoding_errors: 'str | None' = 'strict',
dialect: 'str | csv.Dialect | None' = None, on_bad_lines: 'str' = 'error', delim_whitespace: 'bool | lib.NoDefault'
= <no_default>, low_memory: 'bool' = True, memory_map: 'bool' = False, float_precision: "Literal['high', 'legacy'] |
None" = None, storage_options: 'StorageOptions | None' = None, dtype_backend: 'DtypeBackend | lib.NoDefault' = <no_d
efault>) -> 'DataFrame | TextFileReader'
    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 <https://pandas.pydata.org/pandas-docs/stable/user_guide/io.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, gs, and file. For file URLs, a host is
        expected. A local file could be: file://localhost/path/to/table.csv.

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

        By file-like object, we refer to objects with a ``read()`` method, such as
        a file handle (e.g. via builtin ``open`` function) or ``StringIO``.
    sep : str, default ','
        Character or regex pattern to treat as the delimiter. If ``sep=None``, the
        C engine cannot automatically detect
        the separator, but the Python parsing engine can, meaning the latter will
        be used and automatically detect the separator from only the first valid
        row of the file by Python's builtin sniffer tool, ``csv.Sniffer``.
        In addition, separators longer than 1 character and different from
        ``'\s+'`` will be interpreted as regular expressions 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, optional
        Alias for ``sep``.
    header : int, Sequence of int, 'infer' or None, default 'infer'
        Row number(s) containing column labels and marking the start of the
        data (zero-indexed). Default behavior is to infer the column names: if no ``names``
        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 to ``names`` 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 that
        specify row locations for a :class:`~pandas.MultiIndex` 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 line of
        data rather than the first line of the file.
    names : Sequence of Hashable, optional
        Sequence of column labels to apply. If the file contains a header row,
        then you should explicitly pass ``header=0`` to override the column names.
        Duplicates in this list are not allowed.
    index_col : Hashable, Sequence of Hashable or False, optional
      Column(s) to use as row label(s), denoted either by column labels or column
      indices.  If a sequence of labels or indices is given, :class:`~pandas.MultiIndex`
      will be formed for the row labels.

      Note: ``index_col=False`` can be used to force pandas to *not* use the first
      column as the index, e.g., when you have a malformed file with delimiters at
      the end of each line.
    usecols : Sequence of Hashable or Callable, optional
        Subset of columns to select, denoted either by column labels or column indices.
        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). If ``names`` are given, the document
        header row(s) are not taken into account. 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 :class:`~pandas.DataFrame` from ``data`` with element order
    preserved use ``pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]``
    for 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 the column
    names, returning names where the callable function evaluates to ``True``. An
    example of a valid callable argument would be ``lambda x: x.upper() in
    ['AAA', 'BBB', 'DDD']``. Using this parameter results in much faster
    parsing time and lower memory usage.
dtype : dtype or dict of {Hashable : dtype}, optional
    Data type(s) to apply to either the whole dataset or individual columns.
    E.g., ``{'a': np.float64, 'b': np.int32, 'c': 'Int64'}``
    Use ``str`` or ``object`` together with suitable ``na_values`` settings
    to preserve and not interpret ``dtype``.
    If ``converters`` are specified, they will be applied INSTEAD
    of ``dtype`` conversion.

    .. versionadded:: 1.5.0

        Support for ``defaultdict`` was added. Specify a ``defaultdict`` as input where
        the default determines the ``dtype`` of the columns which are not explicitly
        listed.
engine : {'c', 'python', 'pyarrow'}, optional
    Parser engine to use. The C and pyarrow engines are faster, while the python engine
    is currently more feature-complete. Multithreading is currently only supported by
    the pyarrow engine.

    .. versionadded:: 1.4.0

        The 'pyarrow' engine was added as an *experimental* engine, and some features
        are unsupported, or may not work correctly, with this engine.
converters : dict of {Hashable : Callable}, optional
    Functions for converting values in specified columns. Keys can either
    be column labels or column indices.
true_values : list, optional
    Values to consider as ``True`` in addition to case-insensitive variants of 'True'.
false_values : list, optional
    Values to consider as ``False`` in addition to case-insensitive variants of 'False'.
skipinitialspace : bool, default False
    Skip spaces after delimiter.
skiprows : int, list of int or Callable, optional
    Line numbers to skip (0-indexed) or number of lines to skip (``int``)
    at the start of the file.

    If callable, the callable function will be evaluated against the 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 ``engine='c'``).
nrows : int, optional
    Number of rows of file to read. Useful for reading pieces of large files.
na_values : Hashable, Iterable of Hashable or dict of {Hashable : Iterable}, optional
    Additional strings to recognize as ``NA``/``NaN``. If ``dict`` passed, specific
    per-column ``NA`` values.  By default the following values are interpreted as
    ``NaN``: " ", "#N/A", "#N/A N/A", "#NA", "-1.#IND", "-1.#QNAN", "-NaN", "-nan",
    "1.#IND", "1.#QNAN", "<NA>", "N/A", "NA", "NULL", "NaN", "None",
    "n/a", "nan", "null ".
keep_default_na : bool, default True
    Whether or not to include the default ``NaN`` values when parsing the 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 specified, 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 specified, no
      strings will be parsed as ``NaN``.

    Note that if ``na_filter`` is passed in as ``False``, the ``keep_default_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 ``NA`` values, passing ``na_filter=False`` can improve the
    performance of reading a large file.
verbose : bool, default False
    Indicate number of ``NA`` values placed in non-numeric columns.

    .. deprecated:: 2.2.0
skip_blank_lines : bool, default True
    If ``True``, skip over blank lines rather than interpreting as ``NaN`` values.
parse_dates : bool, list of Hashable, list of lists or dict of {Hashable : list}, default False
    The behavior is as follows:
```

```
    * ``bool``. If ``True`` -> try parsing the index. Note: Automatically set to
      ``True`` if ``date_format`` or ``date_parser`` arguments have been passed.
    * ``list`` of ``int`` or names. e.g. If ``[1, 2, 3]`` -> try parsing columns 1, 2, 3
      each as a separate date column.
    * ``list`` of ``list``. e.g.  If ``[[1, 3]]`` -> combine columns 1 and 3 and parse
      as a single date column. Values are joined with a space before parsing.
    * ``dict``, e.g. ``{'foo' : [1, 3]}`` -> parse columns 1, 3 as date and call
      result 'foo'. Values are joined with a space before parsing.

    If a column or index cannot be represented as an array of ``datetime``,
    say because of an unparsable 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 :func:`~pandas.to_datetime` after
    :func:`~pandas.read_csv`.

    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 infer 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 can increase
    the parsing speed by 5-10x.

    .. deprecated:: 2.0.0
        A strict version of this argument is now the default, passing it has no effect.

keep_date_col : bool, default False
    If ``True`` and ``parse_dates`` specifies combining multiple columns then
    keep the original columns.
date_parser : Callable, 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 different ways,
    advancing to the next if an exception occurs: 1) Pass one or more 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 using one or
    more strings (corresponding to the columns defined by ``parse_dates``) as
    arguments.

    .. deprecated:: 2.0.0
       Use ``date_format`` instead, or read in as ``object`` and then apply
       :func:`~pandas.to_datetime` as-needed.
date_format : str or dict of column -> format, optional
    Format to use for parsing dates when used in conjunction with ``parse_dates``.
    The strftime to parse time, e.g. :const:`"%d/%m/%Y"`. See
    `strftime documentation
    <https://docs.python.org/3/library/datetime.html
    #strftime-and-strptime-behavior>`_ for more information on choices, though
    note that :const:`"%f"` will parse all the way up to nanoseconds.
    You can also pass:

    - "ISO8601", to parse any `ISO8601 <https://en.wikipedia.org/wiki/ISO_8601>`_
        time string (not necessarily in exactly the same format);
    - "mixed", to infer the format for each element individually. This is risky,
        and you should probably use it along with `dayfirst`.

    .. versionadded:: 2.0.0
dayfirst : bool, default False
    DD/MM format dates, international and European format.
cache_dates : bool, default True
    If ``True``, use a cache of unique, converted dates to apply the ``datetime``
    conversion. May produce significant speed-up when parsing duplicate
    date strings, especially ones with timezone offsets.

iterator : bool, default False
    Return ``TextFileReader`` object for iteration or getting chunks with
    ``get_chunk()``.
chunksize : int, optional
    Number of lines to read from the file per chunk. Passing a value will cause the
    function to return a ``TextFileReader`` object for iteration.
    See the `IO Tools docs
    <https://pandas.pydata.org/pandas-docs/stable/io.html#io-chunking>`_
    for more information on ``iterator`` and ``chunksize``.

compression : str or dict, 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', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2'
    (otherwise no compression).
    If using 'zip' or 'tar', the ZIP file must contain only one data file to be read in.
    Set to ``None`` for no decompression.
    Can also be a dict with key ``'method'`` set
    to one of {``'zip'``, ``'gzip'``, ``'bz2'``, ``'zstd'``, ``'xz'``, ``'tar'``} and
    other key-value pairs are forwarded to
    ``zipfile.ZipFile``, ``gzip.GzipFile``,
    ``bz2.BZ2File``, ``zstandard.ZstdDecompressor``, ``lzma.LZMAFile`` or
```

```
        ``tarfile.TarFile``, respectively.
        As an example, the following could be passed for Zstandard decompression using a
        custom compression dictionary:
        ``compression={'method': 'zstd', 'dict_data': my_compression_dict}``.

        .. versionadded:: 1.5.0
            Added support for `.tar` files.

        .. versionchanged:: 1.4.0 Zstandard support.

    thousands : str (length 1), optional
        Character acting as the thousands separator in numerical values.
    decimal : str (length 1), default '.'
        Character to recognize as decimal point (e.g., use ',' for European data).
    lineterminator : str (length 1), optional
        Character used to denote a line break. Only valid with C parser.
    quotechar : str (length 1), optional
        Character used to denote the start and end of a quoted item. Quoted
        items can include the ``delimiter`` and it will be ignored.
    quoting : {0 or csv.QUOTE_MINIMAL, 1 or csv.QUOTE_ALL, 2 or csv.QUOTE_NONNUMERIC, 3 or csv.QUOTE_NONE}, default
csv.QUOTE_MINIMAL
        Control field quoting behavior per ``csv.QUOTE_*`` constants. Default is
        ``csv.QUOTE_MINIMAL`` (i.e., 0) which implies that only fields containing special
        characters are quoted (e.g., characters defined in ``quotechar``, ``delimiter``,
        or ``lineterminator``.
    doublequote : bool, default True
       When ``quotechar`` is specified and ``quoting`` is not ``QUOTE_NONE``, indicate
       whether or not to interpret two consecutive ``quotechar`` elements INSIDE a
       field as a single ``quotechar`` element.
    escapechar : str (length 1), optional
        Character used to escape other characters.
    comment : str (length 1), optional
        Character indicating that the remainder of line should not be parsed.
        If found at the 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_lines=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, default 'utf-8'
        Encoding to use for UTF when reading/writing (ex. ``'utf-8'``). `List of Python
        standard encodings
        <https://docs.python.org/3/library/codecs.html#standard-encodings>`_ .

    encoding_errors : str, optional, default 'strict'
        How encoding errors are treated. `List of possible values
        <https://docs.python.org/3/library/codecs.html#error-handlers>`_ .

        .. versionadded:: 1.3.0

    dialect : str or csv.Dialect, optional
        If provided, this parameter will override values (default or not) for the
        following parameters: ``delimiter``, ``doublequote``, ``escapechar``,
        ``skipinitialspace``, ``quotechar``, and ``quoting``. If it is necessary to
        override values, a ``ParserWarning`` will be issued. See ``csv.Dialect``
        documentation for more details.
    on_bad_lines : {'error', 'warn', 'skip'} or Callable, default 'error'
        Specifies what to do upon encountering a bad line (a line with too many fields).
        Allowed values are :

        - ``'error'``, raise an Exception when a bad line is encountered.
        - ``'warn'``, raise a warning when a bad line is encountered and skip that line.
        - ``'skip'``, skip bad lines without raising or warning when they are encountered.

        .. versionadded:: 1.3.0

        .. versionadded:: 1.4.0

            - Callable, function with signature
              ``(bad_line: list[str]) -> list[str] | None`` that will process a single
              bad line. ``bad_line`` is a list of strings split by the ``sep``.
              If the function returns ``None``, the bad line will be ignored.
              If the function returns a new ``list`` of strings with more elements than
              expected, a ``ParserWarning`` will be emitted while dropping extra elements.
              Only supported when ``engine='python'``

        .. versionchanged:: 2.2.0

            - Callable, function with signature
              as described in `pyarrow documentation
              <https://arrow.apache.org/docs/python/generated/pyarrow.csv.ParseOptions.html
              #pyarrow.csv.ParseOptions.invalid_row_handler>`_ when ``engine='pyarrow'``

    delim_whitespace : bool, default False
        Specifies whether or not whitespace (e.g. ``' '`` or ``'\t'``) will be
        used as the ``sep`` delimiter. Equivalent to setting ``sep='\s+'``. If this option
```

```
    is set to ``True``, nothing should be passed in for the ``delimiter``
    parameter.

    .. deprecated:: 2.2.0
        Use ``sep="\s+"`` instead.
low_memory : bool, default True
    Internally process the file in chunks, resulting in lower memory use
    while parsing, but possibly mixed type inference.  To ensure no mixed
    types either set ``False``, or specify the type with the ``dtype`` parameter.
    Note that the entire file is read into a single :class:`~pandas.DataFrame`
    regardless, 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 file object
    directly onto memory and access the data directly from there. Using this
    option can improve performance because there is no longer any I/O overhead.
float_precision : {'high', 'legacy', 'round_trip'}, optional
    Specifies which converter the C engine should use for floating-point
    values. The options are ``None`` or ``'high'`` for the ordinary converter,
    ``'legacy'`` for the original lower precision pandas converter, and
    ``'round_trip'`` for the round-trip converter.

storage_options : dict, optional
    Extra options that make sense for a particular storage connection, e.g.
    host, port, username, password, etc. For HTTP(S) URLs the key-value pairs
    are forwarded to ``urllib.request.Request`` as header options. For other
    URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are
    forwarded to ``fsspec.open``. Please see ``fsspec`` and ``urllib`` for more
    details, and for more examples on storage options refer `here
    <https://pandas.pydata.org/docs/user_guide/io.html?
    highlight=storage_options#reading-writing-remote-files>`_.

dtype_backend : {'numpy_nullable', 'pyarrow'}, default 'numpy_nullable'
    Back-end data type applied to the resultant :class:`DataFrame`
    (still experimental). Behaviour is as follows:

    * ``"numpy_nullable"``: returns nullable-dtype-backed :class:`DataFrame`
      (default).
    * ``"pyarrow"``: returns pyarrow-backed nullable :class:`ArrowDtype`
      DataFrame.

    .. versionadded:: 2.0

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

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

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 [15]:
```
df = pd.read_csv('data/adult_test.csv')
```

Data transformations: pandas data frames

By the end of this lecture, 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 [4]:
```
# df.iloc[] - for more info, see https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#indexing-int
# 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])
```

```
age                             44
workclass                  Private
fnlwgt                       83891
education                Bachelors
education-num                   13
marital-status            Divorced
occupation             Adm-clerical
relationship              Own-child
race              Asian-Pac-Islander
sex                             Male
capital-gain                    5455
capital-loss                       0
hours-per-week                    40
native-country         United-States
gross-income                  <=50K.
Name: 16279, dtype: object
```

In [5]:
```
# 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
```

```
   age workclass  fnlwgt      education  education-num       marital-status  \
1   38   Private   89814        HS-grad              9   Married-civ-spouse
4   18         ?  103497   Some-college             10        Never-married
9   55   Private  104996        7th-8th              4   Married-civ-spouse

         occupation relationship    race      sex  capital-gain  capital-loss  \
1   Farming-fishing      Husband   White     Male             0             0
4                 ?    Own-child   White   Female             0             0
9      Craft-repair      Husband   White     Male             0             0

   hours-per-week  native-country gross-income
1              50   United-States       <=50K.
4              30   United-States       <=50K.
9              10   United-States       <=50K.
```

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

In [6]:
```
# df.loc[] - for more info, see https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#indexing-labe

#print(df.index) # the default index when reading in a file is a range index. 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)
```

```
Help on method set_index in module pandas.core.frame:

set_index(keys, *, drop: 'bool' = True, append: 'bool' = False, inplace: 'bool' = False, verify_integrity: 'bool' =
False) -> 'DataFrame | None' method of pandas.core.frame.DataFrame instance
    Set the DataFrame index using existing columns.

    Set the DataFrame index (row labels) using one or more existing
    columns or arrays (of the correct length). The index can replace the
    existing index or expand on it.

    Parameters
    ----------
    keys : label or array-like or list of labels/arrays
        This parameter can be either a single column key, a single array of
        the same length as the calling DataFrame, or a list containing an
        arbitrary combination of column keys and arrays. Here, "array"
        encompasses :class:`Series`, :class:`Index`, ``np.ndarray``, and
        instances of :class:`~collections.abc.Iterator`.
    drop : bool, default True
        Delete columns to be used as the new index.
    append : bool, default False
        Whether to append columns to existing index.
    inplace : bool, default False
        Whether to modify the DataFrame rather than creating a new one.
    verify_integrity : bool, default False
        Check the new index for duplicates. Otherwise defer the check until
        necessary. Setting to False will improve the performance of this
        method.

    Returns
    -------
    DataFrame or None
        Changed row labels or None if ``inplace=True``.

    See Also
    --------
    DataFrame.reset_index : Opposite of set_index.
    DataFrame.reindex : Change to new indices or expand indices.
    DataFrame.reindex_like : Change to same indices as other DataFrame.

    Examples
    --------
    >>> df = pd.DataFrame({'month': [1, 4, 7, 10],
    ...                    'year': [2012, 2014, 2013, 2014],
    ...                    'sale': [55, 40, 84, 31]})
    >>> df
       month  year  sale
    0      1  2012    55
    1      4  2014    40
    2      7  2013    84
    3     10  2014    31

    Set the index to become the 'month' column:

    >>> df.set_index('month')
           year  sale
    month
    1      2012    55
    4      2014    40
    7      2013    84
    10     2014    31

    Create a MultiIndex using columns 'year' and 'month':

    >>> df.set_index(['year', 'month'])
                sale
    year  month
    2012  1     55
    2014  4     40
    2013  7     84
    2014  10    31

    Create a MultiIndex using an Index and a column:

    >>> df.set_index([pd.Index([1, 2, 3, 4]), 'year'])
             month  sale
       year
    1  2012  1      55
    2  2014  4      40
    3  2013  7      84
    4  2014  10     31

    Create a MultiIndex using two Series:

    >>> s = pd.Series([1, 2, 3, 4])
    >>> df.set_index([s, s**2])
          month  year  sale
```

```
   1 1       1  2012    55
   2 4       4  2014    40
   3 9       7  2013    84
   4 16     10  2014    31
```

In [7]:
```
df_index_age = df.set_index('age',drop=False)

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

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

```
     age   workclass  fnlwgt    education  education-num       marital-status  \
age
30    30     Private  101135    Bachelors             13        Never-married
30    30     Private  229636      HS-grad              9   Married-civ-spouse
30    30     Private  142921   Assoc-acdm             12        Never-married
30    30   State-gov  260782      HS-grad              9        Never-married
30    30     Private  296462      HS-grad              9        Never-married

              occupation    relationship    race      sex  capital-gain  \
age
30      Exec-managerial   Not-in-family   White   Female             0
30    Machine-op-inspct         Husband   White     Male             0
30       Prof-specialty   Not-in-family   White   Female             0
30        Other-service   Not-in-family   White     Male             0
30      Exec-managerial   Not-in-family   Black     Male             0

     capital-loss  hours-per-week  native-country gross-income
age
30              0              50   United-States       <=50K.
30              0              40          Mexico       <=50K.
30              0              40   United-States       <=50K.
30              0              40   United-States       <=50K.
30              0              40   United-States       <=50K.
```

3) select rows based on column condition

In [8]:
```
# one condition
#print(df[df['age']==30].head())
# here is the condition: it's a boolean series - series is basically a dataframe 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')])
```

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	gross-income
899	90	Private	149069	Assoc-acdm	12	Married-civ-spouse	Sales	Husband	White	Male	0	1825	50	United-States	>50K.
2047	65	Private	444725	Prof-school	15	Married-spouse-absent	Craft-repair	Not-in-family	White	Male	0	0	48	Hungary	>50K.
2779	55	Self-emp-not-inc	218456	Masters	14	Divorced	Exec-managerial	Not-in-family	White	Female	0	0	50	Hungary	<=50K.
3496	90	Self-emp-not-inc	83601	Prof-school	15	Widowed	Prof-specialty	Not-in-family	White	Male	1086	0	60	United-States	<=50K.
6822	44	Private	254303	Masters	14	Never-married	Prof-specialty	Not-in-family	White	Male	0	0	40	Hungary	>50K.
6976	90	Private	250832	HS-grad	9	Married-civ-spouse	Transport-moving	Husband	White	Male	2414	0	40	United-States	<=50K.
7414	90	Private	227796	Assoc-acdm	12	Never-married	Exec-managerial	Not-in-family	White	Male	6097	0	45	United-States	>50K.
7419	90	Self-emp-not-inc	122348	Prof-school	15	Divorced	Prof-specialty	Not-in-family	White	Male	20051	0	45	United-States	>50K.
8427	90	Federal-gov	311184	Masters	14	Married-civ-spouse	Craft-repair	Husband	White	Male	0	0	99	United-States	<=50K.
8982	90	Private	225063	HS-grad	9	Married-civ-spouse	Craft-repair	Husband	Asian-Pac-Islander	Male	0	0	40	South	<=50K.
10666	71	?	158437	5th-6th	3	Married-civ-spouse	?	Husband	White	Male	0	0	40	Hungary	<=50K.
10735	90	Local-gov	188242	HS-grad	9	Never-married	Craft-repair	Own-child	White	Male	11678	0	40	United-States	>50K.
11871	90	?	50746	10th	6	Divorced	?	Not-in-family	White	Female	0	0	7	United-States	<=50K.
12437	53	Self-emp-not-inc	169112	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	40	Hungary	>50K.
12446	90	Private	347074	Some-college	10	Never-married	Adm-clerical	Own-child	White	Female	0	1944	12	United-States	<=50K.
13958	90	Private	272752	Some-college	10	Never-married	Other-service	Own-child	White	Male	0	0	10	United-States	<=50K.
15088	90	Private	197613	HS-grad	9	Never-married	Adm-clerical	Not-in-family	White	Female	0	0	40	United-States	>50K.
15404	27	Self-emp-not-inc	177831	HS-grad	9	Married-civ-spouse	Craft-repair	Husband	White	Male	0	0	40	Hungary	<=50K.

Exercise 2

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

In [ ]:

Data transformations: pandas data frames

By the end of this lecture, 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 [10]:
```
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])
```

```
       age  fnlwgt  education-num         occupation    race  capital-gain  \
0       39   77516             13       Adm-clerical   White          2174
1       50   83311             13    Exec-managerial   White             0
2       38  215646              9  Handlers-cleaners   White             0
3       53  234721              7  Handlers-cleaners   Black             0
4       28  338409             13     Prof-specialty   Black             0
...    ...     ...            ...                ...     ...           ...
32556   27  257302             12       Tech-support   White             0
32557   40  154374              9  Machine-op-inspct   White             0
32558   58  151910              9       Adm-clerical   White             0
32559   22  201490              9       Adm-clerical   White             0
32560   52  287927              9    Exec-managerial   White         15024

       hours-per-week gross-income
0                  40        <=50K
1                  13        <=50K
2                  40        <=50K
3                  40        <=50K
4                  40        <=50K
...               ...          ...
32556              38        <=50K
32557              40         >50K
32558              40        <=50K
32559              20        <=50K
32560              40         >50K

[32561 rows x 8 columns]
```


Data transformations: pandas data frames

By the end of this lecture, 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 [11]:
```
# 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
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 [12]:
```
# we are interested in only patients from hospital1
df_left = df1.merge(df2,how='left',on='ID') # IDs from the left dataframe (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 dataframe (df2) are kept
#df_right = df2.merge(df1,how='left',on='ID')
#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 hospitals
df_outer = df1.merge(df2,how='outer',on='ID')  # merging on IDs present in any dataframe
print(df_outer)
```

```
    ID  col1 col2_x  col3 col2_y
0  ID1     5      y   NaN    NaN
1  ID2     8      j  12.0      q
2  ID3     2      w   NaN    NaN
3  ID4     6      b   NaN    NaN
4  ID5     0      a  76.0      u
5  ID6     2      b  34.0      e
6  ID7     5      t   NaN    NaN
     ID  col1 col2_x  col3 col2_y
0   ID1   5.0      y   NaN    NaN
1  ID10   NaN    NaN  98.0      l
2  ID11   NaN    NaN  65.0      p
3   ID2   8.0      j  12.0      q
4   ID3   2.0      w   NaN    NaN
5   ID4   6.0      b   NaN    NaN
6   ID5   0.0      a  76.0      u
7   ID6   2.0      b  34.0      e
8   ID7   5.0      t   NaN    NaN
```

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 [13]:
```
#df_append = pd.concat([df1,df2]) # note that rows with ID2, ID5, and ID6  are duplicated! Indices are duplicated t
#print(df_append)

df_append = pd.concat([df1,df2],ignore_index=True) # note that rows with ID2, ID5, and ID6  are duplicated!
print(df_append)

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

# df_append = pd.concat([df1,df2,df3],ignore_index=True) # multiple dataframes can be appended
# 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
7    ID2   NaN    q  12.0
8    ID5   NaN    u  76.0
9    ID6   NaN    e  34.0
10  ID10   NaN    l  98.0
11  ID11   NaN    p  65.0
```

Exercise 3

In [14]:
```
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', '11'],
        '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 from df_append are present.
# Assign the new data frame to df_merge.
# How many rows and columns do we have in df_merge?
```

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

Mud card

In [ ]:
```
```