# Python for Machine Learning and Data Analysis

Handan Liu, PhD

h.liu@northeastern.edu

Northeastern University

Spring 2019

# Pandas: Powerful Python Data Analysis Toolkit

Lecture 3

# Preface: NumPy vs. Pandas

- NumPy's ndarray data structure provides essential features for the type of clean, well-organized data typically seen in numerical computing tasks.
- While it serves this purpose very well, its limitations become clear when we need more flexibility (e.g., attaching labels to data, working with missing data, etc.) and when attempting operations that do not map well to element-wise broadcasting (e.g., groupings, pivots, etc.), each of which is an important piece of analyzing the less structured data available in many forms in the world around us.
- Pandas, and in particular its Series and DataFrame objects, builds on the NumPy array structure and provides efficient access to these sorts of "data munging" tasks that occupy much of a data scientist's time.

- As you read about Numpy, don't forget that Jupyter gives you the ability to quickly explore the contents of a package (by using the tab-completion feature) as well as the documentation of various functions (using the ? character).

  Import pandas as pd

  pd.[TAB]

  pd?

# Contents

- Introducing Pandas Objects
  - Series
  - DataFrame
  - Index
- Data Indexing and Selection
- Operations in Pandas
- Handing Missing Data
- Hierarchical Indexing
- Example: Pandas analysis for Titanic case

# Introducing Pandas Objects

- Pandas objects can be thought of as enhanced versions of NumPy structured arrays:
  - the rows and columns are identified with labels rather than simple integer indices (Numpy)
- Three fundamental Pandas data structures:
  - Series
  - DataFrame
  - Index

# Pandas Object: Series

- One-dimensional array of indexed data
- Series wraps both a sequence of values and a sequence of indices
- The values are simply a NumPy array.
- The index is an array-like object of type pd.index
- Pandas data can be accessed by the associated index via Python square-bracket notation.

- Series as generalized NumPy array
  - The essential difference is the presence of the index: while the Numpy Array has an implicitly defined integer index used to access the values, the Pandas Series has an explicitly defined index associated with the values.
  - This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type; even non-contiguous or non-sequential indices:

- Series as specialized dictionary
  - A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a Series is a structure which maps typed keys to a set of typed values.
  - This typing is important: the type information of a Pandas Series makes it much more efficient than Python dictionaries for certain operations.
  - The Series-as-dictionary analogy can be made even more clear by constructing a Series object directly from a Python dictionary.
  - By default, a Series will be created where the index is drawn from the sorted keys. From here, typical dictionary-style item access can be performed.
  - Unlike a dictionary, though, the Series also supports array-style operations such as slicing:

- Constructing Series objects
  - Pandas Series:

  >>> pd.Series (data, index=index)
  - where index is an optional argument, and data can be one of many entities.
  - data can be a list or NumPy array, in which case index defaults to an integer sequence.
  - data can be a scalar, which is repeated to fill the specified index.
  - data can be a dictionary, in which index defaults to the sorted dictionary keys.
  - In each case, the index can be explicitly set if a different result is preferred.

# Pandas Object: DataFrame

- DataFrame as a generalized NumPy array
  - If a Series is an analog of a one-dimensional array with flexible indices, a DataFrame is an analog of a two-dimensional array with both flexible row indices and flexible column names.
  - Just as you might think of a two-dimensional array as an ordered sequence of aligned one-dimensional columns, you can think of a DataFrame as a sequence of aligned Series objects. Here, by "aligned" we mean that they share the same index.
  - Like the Series object, the DataFrame has an index attribute that gives access to the index labels.
  - Additionally, the DataFrame has a columns attribute, which is an Index object holding the column labels.
  - Thus the DataFrame can be thought of as a generalization of a two-dimensional NumPy array, where both the rows and columns have a generalized index for accessing the data.

- DataFrame as specialized dictionary
  - Similarly, we can also think of a DataFrame as a specialization of a dictionary. Where a dictionary maps a key to a value, a DataFrame maps a column name to a Series of column data. For example, asking for the 'area' attribute returns the Series object containing the areas we saw earlier:

- Constructing DataFrame objects
  - From a single Series object:
    - A DataFrame is a collection of Series objects, and a single-column DataFrame can be constructed from a single Series:
  - From a list of dicts
    - Any list of dictionaries can be made into a DataFrame. We'll use a simple list comprehension to create some data:
    - Even if some keys in the dictionary are missing, Pandas will fill them in with NaN (i.e., "not a number") values:
  - From a dictionary of Series objects
    - A DataFrame can be constructed from a dictionary of Series objects as well:
  - From a two-dimensional NumPy array
    - Given a two-dimensional array of data, we can create a DataFrame with any specified column and index names. If omitted, an integer index will be used for each.
  - From a NumPy structured array
    - A DataFrame operates much like a structured array, and can be created directly from one.

# Pandas Object: Index

- Index as immutable array
  - The Index in many ways operates like an array. For example, we can use standard Python indexing notation to retrieve values or slices:
  - Index objects also have many of the attributes familiar from NumPy arrays:
  - One difference between Index objects and NumPy arrays is that indices are immutable–that is, they cannot be modified via the normal means:
  - This immutability makes it safer to share indices between multiple DataFrames and arrays, without the potential for side effects from inadvertent index modification.
- Index as ordered set
  - Pandas objects are designed to facilitate operations such as joins across datasets, which depend on many aspects of set arithmetic.
  - The Index object follows many of the conventions used by Python's built-in set data structure, so that unions, intersections, differences, and other combinations can be computed in a familiar way:

# Data Indexing and Selection:
# Data Selection in Series

- Series as dictionary
  - Like a dictionary, the Series object provides a mapping from a collection of keys to a collection of values.
  - Series objects can even be modified with a dictionary-like syntax. Just as you can extend a dictionary by assigning to a new key, you can extend a Series by assigning to a new index value.
  - This easy mutability of the objects is a convenient feature: under the hood, Pandas is making decisions about memory layout and data copying that might need to take place; the user generally does not need to worry about these issues.
- Series as one-dimensional array
  - A Series builds on this dictionary-like interface and provides array-style item selection via the same basic mechanisms as NumPy arrays – that is, slices, masking, and fancy indexing.
  - Among these, slicing may be the source of the most confusion. Notice that when slicing with an explicit index (i.e., data['a':'c']), the final index is included in the slice, while when slicing with an implicit index (i.e., data[0:2]), the final index is excluded from the slice.

# Cont'd

- Pandas provides some special *indexer* attributes that explicitly expose certain indexing schemes.
  - Indexers*: loc* and *iloc*
  - The *loc* attribute allows indexing and slicing that always references the explicit index.
  - The *iloc* attribute allows indexing and slicing that always references the implicit Python-style index.
- One guiding principle of Python code is that "explicit is better than implicit." The explicit nature of *loc* and *iloc* make them very useful in maintaining clean and readable code; especially in the case of integer indexes.
- Recommend using these both to make code easier to read and understand, and to prevent subtle bugs due to the mixed indexing/slicing convention.

# Data Indexing and Selection: Data Selection in DataFrame

- DataFrame as a dictionary
  - The first analogy is the DataFrame as a dictionary of related Series objects.
  - The individual Series that make up the columns of the DataFrame can be accessed via dictionary-style indexing of the column name.
  - Equivalently, we can use attribute-style access with column names that are strings.
  - This attribute-style column access actually accesses the exact same object as the dictionary-style access:
  - Modify the object:  the straightforward syntax of element-by-element arithmetic between Series objects.

# Cont'd

- DataFrame as two-dimensional array
  - View the DataFrame as an enhanced two-dimensional array:
    - Can not simply treat it as a NumPy array.
  - For array-style indexing, Pandas again uses the loc and iloc indexers.
    - Using the *iloc* indexer: index the underlying array as if it is a simple NumPy array (using the implicit Python-style index).
    - Using the *loc* indexer: index the underlying data in an array-like style but using the explicit index and column names.
    - Any of the familiar NumPy-style data access patterns can be used within these indexers.
    - Any of these indexing conventions may also be used to set or modify values; this is done in the standard way that you might be accustomed to from working with NumPy.

# Cont'd:

- Additional indexing conventions:
  - *indexing* refers to columns;
  - *slicing* refers to rows.

- These two conventions are syntactically similar to those on a NumPy array, and while these may not precisely fit the mold of the Pandas conventions, they are nevertheless quite useful in practice.

# Operating on Data in Pandas

- Ufuncs: Index Preservation
  - Because Pandas is designed to work with NumPy, any NumPy ufunc will work on Pandas Series and DataFrame objects.
  - If we apply a NumPy ufunc on either of these objects, the result will be another Pandas object with the indices preserved.
- UFuncs: Index Alignment
  - Index alignment in Series:
    - Combing two different data sources, the resulting array contains the union of indices of the two input arrays, which could be determined using standard Python set arithmetic on these indices.
    - Any item for which one or the other does not have an entry is marked with NaN, or "Not a Number," which is how Pandas marks missing.
    - This index matching is implemented this way for any of Python's built-in arithmetic expressions; any missing values are filled in with NaN by default.
    - If using NaN values is not the desired behavior, the fill value can be modified using appropriate object methods in place of the operators. For example, calling A.add(B) is equivalent to calling A + B, but allows optional explicit specification of the fill value for any elements in A or B that might be missing.

# Cont’d

- Index alignment in DataFrame
  - A similar type of alignment takes place for both columns and indices when performing operations on DataFrames.
  - Notice that indices are aligned correctly irrespective of their order in the two objects, and indices in the result are sorted. As was the case with Series, we can use the associated object's arithmetic method and pass any desired *fill_value* to be used in place of missing entries.

The following table lists Python operators and their equivalent Pandas object methods:

| Python Operator | Pandas Method(s) |
|---|---|
| + | add() |
| - | sub(), subtract() |
| * | mul(), multiply() |
| / | truediv(), div(), divide() |
| // | floordiv() |
| % | mod() |
| ** | pow() |

# Cont'd

- Ufuncs: Operations Between DataFrame and Series
  - Operations between a DataFrame and a Series are similar to operations between a two-dimensional and one-dimensional NumPy array.
  - In Pandas, the convention similarly operates row-wise by default.
  - If you would instead like to operate column-wise, you can use the object methods mentioned earlier, while specifying the axis keyword.
  - These DataFrame/Series operations will automatically align indices between the two elements.
- This preservation and alignment of indices and columns means that operations on data in Pandas will always maintain the data context, which prevents the types of silly errors that might come up when working with heterogeneous and/or misaligned data in raw NumPy arrays.

# Handling Missing Data

- Why does missing data exist?
  - The difference between data found in many tutorials and data in the real world is that real-world data is rarely clean and homogeneous.
  - In particular, many interesting datasets will have some amount of data missing.
  - Different data sources may indicate missing data in different ways to make matters even more complicated.
- How handling in Pandas?
  - Discuss some general considerations for missing data
  - Discuss how Pandas chooses to represent it, and
  - Demonstrate some built-in Pandas tools for handling missing data in Python.
  - In our examples, we'll refer to missing data in general as *null*, *NaN*, or *NA* values.

# Missing Data in Pandas

- The way in which Pandas handles missing values is constrained by its reliance on the NumPy package, which does not have a built-in notion of NA values for non-floating-point data types.
- With these constraints in mind, Pandas chose to use sentinels for missing data, and further chose to use two already-existing Python null values: the special floating-point NaN value, and the Python None object. This choice has some side effects, as we will see, but in practice ends up being a good compromise in most cases of interest.

- None: Pythonic missing data
  - The first sentinel value used by Pandas is *None*, a Python singleton object that is often used for missing data in Python code.
  - Because it is a Python object, *None* cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type 'object' (i.e., arrays of Python objects):

    vals1 = np.array([1, **None**, 3, 4])

    >>> array([1, None, 3, 4], dtype=object)
  - This *dtype=object* means that the best common type representation NumPy could infer for the contents of the array is that they are Python objects. While this kind of object array is useful for some purposes, any operations on the data will be done at the Python level, with much more overhead than the typically fast operations seen for arrays with native types:
  - The use of Python objects in an array also means that if you perform aggregations like *sum()* or *min()* across an array with a *None* value, you will generally get an error:

- NaN: Missing numerical data
  - *NaN* (abbr. of Not a Number), is different; it is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation:
  - Notice that NumPy chose a native floating-point type for this array: this means that unlike the object array from before, this array supports fast operations pushed into compiled code.
  - Be aware that *NaN* is a bit like a data virus–it infects any other object it touches. Regardless of the operation, the result of arithmetic with *NaN* will be another *NaN*:
  - Aggregates over the values are well defined (i.e., they don't result in an error) but not always useful:
  - NumPy does provide some special aggregations that will ignore these missing values.
  - Keep in mind that *NaN* is specifically a floating-point value; there is no equivalent *NaN* value for integers, strings, or other types.

- NaN and None in Pandas
  - *NaN* and *None* both have their place, and Pandas is built to handle the two of them nearly interchangeably, converting between them where appropriate.
  - For types that don't have an available sentinel value, Pandas automatically type-casts when *NA* values are present. For example, if we set a value in an integer array to *np.nan*, it will automatically be upcast to a floating-point type to accommodate the *NA*.
  - Notice that in addition to casting the integer array to floating point, Pandas automatically converts the None to a *NaN* value.
  - Pandas sentinel/casting approach works quite well in practice and in my experience only rarely causes issues.

The following table lists the upcasting conventions in Pandas when NA values are introduced:

| **Typeclass** | **Conversion When Storing NAs** | **NA Sentinel Value** |
|---|---|---|
| floating | No change | np.nan |
| object | No change | None or np.nan |
| integer | Cast to float64 | np.nan |
| boolean | Cast to object | None or np.nan |

Keep in mind that in Pandas, string data is always stored with an object dtype.

# Operating on Null Values

- Pandas treats *None and NaN* as essentially interchangeable for indicating missing or null values.
- To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:
  - isnull(): Generate a boolean mask indicating missing values
  - notnull(): Opposite of isnull()
  - dropna(): Return a filtered version of the data
  - fillna(): Return a copy of the data with missing values filled or imputed

- Detecting null values: isnull() and notnull()
- Dropping null values:
  - dropna() (which removes NA values) and fillna() (which fills in NA values)
  - For a Series, the result is straightforward.
  - For a DataFrame, there are more options:
    - By default, *dropna()* will drop all rows in which any null value is present.
    - Alternatively, you can drop NA values along a different axis; axis=1 drops all columns containing a null value.
    - The default is *how='any',* such that any row or column (depending on the axis keyword) containing a null value will be dropped. You can also specify *how='all',* which will only drop rows/columns that are all null values.
    - For further control, the *thresh* parameter lets you specify a minimum number of non-null values for the row/column to be kept.

- Filling null values: fillna()
  - Fill NA entries with a single value, such as zero.
  - Specify a forward-fill to propagate the previous value forward.
  - Specify a back-fill to propagate the next values backward.
  - For DataFrames, the options are similar, but we can also specify an axis along which the fills take place.

# Hierarchical Indexing (Multi-indexing)

- Pandas provides hierarchical indexing (also known as multi-indexing) to incorporate multiple index levels within a single index.
- In this way, higher-dimensional data can be compactly represented within the familiar one-dimensional Series and two-dimensional DataFrame objects.
- In this section, we'll explore:
  - the direct creation of MultiIndex objects,
  - considerations when indexing, slicing, and computing statistics across multiply indexed data, and
  - useful routines for converting between simple and hierarchically indexed representations of your data.

- A Multiple Indexed Series
  - Let's start by considering how we might represent two-dimensional data within a one-dimensional Series. For concreteness, we will consider a series of data where each point has a character and numerical key.
- The bad way
  - Suppose you would like to track data about states from two different years. Using the Pandas tools we've already covered, you might be tempted to simply use Python tuples as keys.
  - With this indexing scheme, you can straightforwardly index or slice the series based on this multiple index. But the convenience ends there. For example, if you need to select all values from 2010, you'll need to do some messy (and potentially slow) munging to make it happen:
  - This produces the desired result, but is not as clean (or as efficient for large datasets) as the slicing syntax we've grown to love in Pandas.

- The better way: Pandas MultiIndex

```
# The Better Way: Pandas MultiIndex
index = pd.MultiIndex.from_tuples(index)
index
```

```
                                0         1         2         0     1
MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],
           labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

```
0  (California, 2000)  0  33871648
0  (California, 2010)  1  37253956
1  (New York, 2000)    0  18976457
1  (New York, 2010)    1  19378102
2  (Texas, 2000)       0  20851820
2  (Texas, 2010)       1  25145561
   dtype: int64
```

```
# re-index the series with MultiIndex
pops = pops.reindex(index)
pops
```

```
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

- Reason why we use hierarchical indexing:
  - use multi-indexing to represent two-dimensional data within a one-dimensional Series;
  - use it to represent data of three or more dimensions in a Series or DataFrame.
- MultiIndex as extra dimension
  - The unstack() method will quickly convert a multiple indexed Series into a conventionally indexed DataFrame.
  - Naturally, the stack() method provides the opposite operation.
  - In addition, all the ufuncs and other functionality discussed in Operating on Data in Pandas work with hierarchical indices as well. Here we compute the fraction of people under 18 by year, given the above data.
  - This allows us to easily and quickly manipulate and explore even high-dimensional data.

# Cont'd: Methods of MultiIndex Creation

- The most straightforward way to construct a multiply indexed Series or DataFrame is to simply pass a list of two or more index arrays to the constructor.
- Similarly, to pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a MultiIndex by default.
- Explicit MultiIndex constructors: using the class method constructors available in the pd.MultiIndex:

  Construct the MultiIndex from:

  - a simple list of arrays giving the index values within each level;
  - a list of tuples giving the multiple index values of each point;
  - a Cartesian product of single indices;
  - directly using its internal encoding by passing levels (a list of lists containing available index values for each level) and labels (a list of lists that reference these labels).

- MultiIndex level names:
  - passing the names argument to any of the above MultiIndex constructors, or
  - setting the names attribute of the index after the fact.
  - With more involved datasets, this can be a useful way to keep track of the meaning of various index values.
- MultiIndex for columns
  - In a DataFrame, the rows and columns are completely symmetric;
  - the rows can have multiple levels of indices,
  - the columns can have multiple levels as well.
- For complicated records containing multiple labeled measurements across multiple times for many subjects (people, countries, cities, etc.) use of hierarchical rows and columns can be extremely convenient!

# Cont'd: Indexing and Slicing a MultiIndex

- Multiple indexed Series
  - Single elements can be accessed by indexing with multiple terms.
  - The MultiIndex also supports *partial indexing*, or indexing just one of the levels in the index. The result is another Series, with the lower-level indices maintained.
  - Partial slicing is available as well, as long as the MultiIndex is sorted.
  - With sorted indices, partial indexing can be performed on lower levels by passing an empty slice in the first index.
  - Other types of indexing and selection work as well; for example, selection based on Boolean masks.
  - Selection based on fancy indexing also works.

- Multiply indexed DataFrames
  - A multiple indexed DataFrame behaves in a similar manner.
  - Remember that columns are primary in a DataFrame, and the syntax used for multiply indexed Series applies to the columns.
  - Also, as with the single-index case, we can use the loc and iloc indexers.
  - These indexers provide an array-like view of the underlying two-dimensional data, but each individual index in loc or iloc can be passed a tuple of multiple indices.
  - Working with slices within these index tuples is not especially convenient; trying to create a slice within a tuple will lead to a syntax error:
    - Pandas object: pd.IndexSlice

# Cont'd: Data Aggregations on Multi-Indices

- Pandas has built-in data aggregation methods, such as mean(), sum(), and max().

# Example: Pandas analysis for Titanic case

- Data Overview
  - Read into the data
  - Understand the meaning of each variable
  - View all numeric class variables and their associated statistics
- Simple analysis of data relationships
  - Number of survivors and deaths
  - Distribution of survivors and deaths at each level
  - ……..

# The End!

Assignment: Practice the example code I ran in this class.