Getting started with pandas

Dinh Viet Hoang

Department of Computer Science
Faculty of Information Technology
DATCOM Lab

June 17, 2024

Outline

1. **Introduction**
2. **Essential pandas Concepts**
3. **Conclusion**

Introduction

What is pandas? I

Pandas is a popular open-source library in the Python programming language used for working with tabular data and time series. It provides powerful data structures and analysis tools, making it easy for Python programmers to manipulate, process, and analyze data flexibly and efficiently.

Example:

```
In [1]: import pandas as pd

In [2]: data = {'Name': ['Alice', 'Bob',
    'Charlie', 'David', 'Emily'],
         'Age': [25, 30, 35, 40, 45],
         'Salary': [50000, 60000, 70000, 80000,
            90000]}
```

What is pandas? II

```
In [3]: frame = pd.DataFrame(data)
In [4]: frame
Out[4]:
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2  Charlie   35   70000
3    David   40   80000
4    Emily   45   90000
```

What is pandas? III

```
In [5]: frame.describe()
Out[5]:
                Age          Salary
count      5.000000        5.000000
mean      35.000000   70000.000000
std        8.660254   15811.388301
min       25.000000   50000.000000
25%       30.000000   60000.000000
50%       35.000000   70000.000000
75%       40.000000   80000.000000
max       45.000000   90000.000000
```

Essential pandas Concepts

1. pandas Data Structure

There are some noteworthy data structures in pandas:

- Series
  - A one-dimensional labeled array capable of holding data of various type (integer, float, string, etc.).
  - A quite similar with dictionary
- DataFrame
  - A two-dimensional labeled data structure capable of holding data of various types (integer, float, string, etc.) in a tabular format.
  - We can think of it as a dictionary of series all sharing the same index.
- Index
  - Index is used to represent the labels of the data. Each DataFrame and Series comes with an index, which make it easier for accessing and organizing data.

1.1 pandas Series I

- A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) of the same type and an associated array of data labels, called its index.
- to initializing a Series, use the following constructor:

```
pandas.Series(<data set>, index = <index set>)
```

1.1 pandas Series II

Example:

```
In [6]: s = pd.Series([1,2,3,4,5],
    ['a','b','c','d','e'])
In [7]: s
Out[7]:
a    1
b    2
c    3
d    4
e    5
dtype: int64
```

- if index does not specify, a default one consisting of the integer 0 through (length of data set) - 1 will be create.

1.2 Pandas DataFrame I

- A DataFrame represents a rectangular table of data and contains an ordered, named collection of columns, each of which can be a different value type (numeric, string, Boolean, etc.).
- to initializing a data frame, we can use the following constructor:

```
pd.DataFrame(<dictionary of data set>, <index set>)
```

1.2 Pandas DataFrame II

Example:

```
In [8]: data = {'Name': ['Alice', 'Bob',
   'Charlie', 'David', 'Emily'], 'Age': [25,
   30, 35, 40, 45], 'Salary': [50000, 60000,
   70000, 80000, 90000]}
In [9]: frame = pd.DataFrame(data)
In [10]: frame
Out[10]:
      Name   Age   Salary
0    Alice    25    50000
1      Bob    30    60000
2  Charlie    35    70000
3    David    40    80000
4    Emily    45    90000
```

2. Essential Functionality

This section will walk you through the fundamental mechanics of interacting with the data contained in a Series or DataFrame:

- Reindexing
- Dropping entries from an axis
- Indexing, Selection and Filtering
- Arithmetic and Data Alignment
- Function Application and Mapping
- Sorting and ranking
- Axis indexes with Duplicated Labels

2.1 Reindexing

An important method on pandas objects is reindex, which means to create a new object with the values rearranged to align with the new index.

Calling reindex on Series rearranges the data according to the new index, introducing missing values if any index values were not already present:

```
In [11]: s.reindex(['e', 'd', 'c', 'a', 'm'])
Out[11]:
e      5
d      4
c      3
a      1
m    NaN
dtype: float64
```

2.1 Reindexing

With DataFrame, reindex can alter the (row) index, columns, or both. When passed only a sequence, it reindexes the rows in the result:

```
In [12]: frame = pd.DataFrame({1 : [1,2,3], 2:
    [4,5,6], 3 : [7,8,9]}, index = ['a','b','c'])
In [13]: frame
Out[13]:
   1   2   3
a  1   4   7
b  2   5   8
c  3   6   9
In [13]: frame.reindex(columns = [2,1,3])
Out[13]:
   2   1   3
a  4   1   7
b  5   2   8
c  6   3   9
```

2.1 Reindexing

Table 5-3. `reindex` function arguments

Argument	Description
`labels`	New sequence to use as an index. Can be Index instance or any other sequence-like Python data structure. An Index will be used exactly as is without any copying.
`index`	Use the passed sequence as the new index labels.
`columns`	Use the passed sequence as the new column labels.
`axis`	The axis to reindex, whether `"index"` (rows) or `"columns"`. The default is `"index"`. You can alternately do `reindex(index=new_labels)` or `reindex(columns=new_labels)`.
`method`	Interpolation (fill) method; `"ffill"` fills forward, while `"bfill"` fills backward.
`fill_value`	Substitute value to use when introducing missing data by reindexing. Use `fill_value="missing"` (the default behavior) when you want absent labels to have null values in the result.
`limit`	When forward filling or backfilling, the maximum size gap (in number of elements) to fill.
`tolerance`	When forward filling or backfilling, the maximum size gap (in absolute numeric distance) to fill for inexact matches.
`level`	Match simple Index on level of MultiIndex; otherwise select subset of.
`copy`	If `True`, always copy underlying data even if the new index is equivalent to the old index; if `False`, do not copy the data when the indexes are equivalent.

2.2 Dropping entries from an axis

The drop method will return a new object with the indicated value or values deleted from an axis:

```
In [14]: s1 = s.drop(['a', 'b'])
In [15]: s1
Out[15]:
c    3
d    4
e    5
dtype: int64
```

2.2 Dropping entries from an axis

With DataFrame, index values can be deleted from either axis. Calling drop with a sequence of labels will drop values from the row labels (axis 0):

```
In [16]: frame1 = frame.drop('b')
In [17]: frame1
Out[17]:
   1  2  3
a  1  4  7
c  3  6  9
```

2.2 Dropping entries from an axis

To drop labels from the columns, instead use the columns keyword, or by passing axis = 1/ axis = 'columns':

```
In [18]: f1 = frame.drop(columns = 1)
         #equal to: f1 = frame.drop(1, axis = 1)
In [19]: f1
Out[19]:
   2   3
a  4   7
b  5   8
c  6   9
```

2.3 Indexing, Selection and Filtering

Series indexing (obj[...]) works analogously to NumPy array indexing, except you can use the Series's index values instead of only integers.

Regular []-based indexing will treat integers as labels if the index contains integers, so the behavior differs depending on the data type of the index.

So the preferred way to select index values is with the special loc operator for label index, and iloc operator for integer index:

```
In [20]: s.loc['a':'c']
         #equal to: s.iloc[0 : 3]
Out[20]:
a    1
b    2
c    3
```

You can slide with labels, but it will contain the endpoint

2.3 Indexing, Selection and Filtering

Indexing into a DataFrame retrieves one or more columns either with a single value or sequence.

Like Series, DataFrame has special attributes loc and iloc for label-based and integer-based indexing, respectively. We should use these instead of regular []-base indexing, the reason is the same as with Series.

```
In [21]: frame.loc[['a','c']]
Out[21]:
   1  2  3
a  1  4  7
c  3  6  9
```

Note

Indexing into a DataFrame retrieves columns, but by default, loc and iloc return rows.

2.3 Indexing, Selection and Filtering

Table 5-4. Indexing options with DataFrame

Type	Notes
`df[column]`	Select single column or sequence of columns from the DataFrame; special case conveniences: Boolean array (filter rows), slice (slice rows), or Boolean DataFrame (set values based on some criterion)
`df.loc[rows]`	Select single row or subset of rows from the DataFrame by label
`df.loc[:, cols]`	Select single column or subset of columns by label
`df.loc[rows, cols]`	Select both row(s) and column(s) by label
`df.iloc[rows]`	Select single row or subset of rows from the DataFrame by integer position
`df.iloc[:, cols]`	Select single column or subset of columns by integer position
`df.iloc[rows, cols]`	Select both row(s) and column(s) by integer position
`df.at[row, col]`	Select a single scalar value by row and column label
`df.iat[row, col]`	Select a single scalar value by row and column position (integers)
`reindex` method	Select either rows or columns by labels

2.3 Indexing, Selection and Filtering

Another use case is indexing with a Boolean DataFrame, such as one produced by a scalar comparison. Consider a DataFrame with all Boolean values produced by comparing with a scalar value:

```
In [22]: frame < 5
Out[22]:
       1      2      3
a   True   True  False
b   True  False  False
c   True  False  False
```

2.3 Indexing, Selection and Filtering

We can use this DataFrame to assign the value we want, like 0, to each location with the value True, like so:

```
In [23]: frame2 = frame.copy()
In [24]: frame2.loc[frame2 < 5] = 0
Out[24]:
   1  2  3
a  0  0  7
b  0  5  8
c  0  6  9
```

2.4 Arithmetic and Data Alignment

when you add objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs.

```
In [25]: s1 = pd.Series([1, 2, 3], index = ['a',
    'b', 'c'])
In [26]: s2 = pd.Series([1, 2, 6], index = ['a',
    'b', 'd'])
In [27]: s1 + s2
Out[27]:
a    2.0
b    4.0
c    NaN
d    NaN
dtype: float64
```

2.4 Arithmetic and Data Alignment

In arithmetic operations between differently indexed objects, you might want to fill with a special value, like 0, when an axis label is found in one object but not the other, you can use arithmetic method, with `fill_value` argument.

```
In [28]: s1.add(s2, fill_value = 0)
Out[28]:
a    2.0
b    4.0
c    3.0
d    6.0
```

See Table 5-5 for a listing of Series and DataFrame methods for arithmetic.

2.4 Arithmetic and Data Alignment

Table 5-5. Flexible arithmetic methods

Method	Description
`add, radd`	Methods for addition (+)
`sub, rsub`	Methods for subtraction (-)
`div, rdiv`	Methods for division (/)
`floordiv, rfloordiv`	Methods for floor division (//)
`mul, rmul`	Methods for multiplication (*)
`pow, rpow`	Methods for exponentiation (**)

2.4 Arithmetic and Data Alignment

Operations between DataFrame and Series

By default, arithmetic between DataFrame and Series matches the index of the Series on the columns of the DataFrame, broadcasting down the rows:

```
In [29]: frame = pd.DataFrame({'a': [1,2,3],
   'b': [4,5,6], 'c': [7,8,9]}, index = [1, 2,
   3])
In [30]: frame + s2
Out[30]:
    a   b   c   d
1   2.0 6.0 NaN NaN
2   3.0 7.0 NaN NaN
3   4.0 8.0 NaN NaN
```

2.5 Function Application and Mapping

NumPy ufuncs (element-wise array methods) also work with pandas objects:

```
In [31]: np.log(frame)
Out[31]:
          a         b         c
1  0.000000  1.386294  1.945910
2  0.693147  1.609438  2.079442
3  1.098612  1.791759  2.197225
```

Another frequent operation is applying a function on one-dimensional arrays to each column or row. DataFrame's `apply()` method does exactly this:

```
In [32]: def f(x):
             return x.max() - x.min()
```

2.5 Function Application and Mapping

```
In [33]: frame.apply(f)
Out[32]:
a    2
b    2
c    2
dtype: int64
```

Here the function f, which computes the difference between the maximum and minimum of a Series, is invoked once on each column in frame. The result is a Series having the columns of frame as its index.

If you pass axis="columns" to apply, the function will be invoked once per row instead (apply across the columns).

Use `applymap()` method instead if you want to applly a function to each element of the DataFrame.

2.6 Sorting and ranking

Sorting a dataset by some criterion is another important built-in operation. To sort lexicographically by row or column label, use the `sort_index` method, which returns a new, sorted object:

```
In [33]: obj = pd.Series(np.arange(4),
    index=["d", "a", "b", "c"])
In [34]: obj.sort_index()
Out[34]:
Out[236]:
a    1
b    2
c    3
d    0
dtype: int64
```

2.6 Sorting and ranking

With a DataFrame, we can sort by index on either axis:

```
In [35]: frame =
    pd.DataFrame(np.arange(8).reshape((2, 4))
index=["three", "one"], columns=["d", "a", "b",
    "c"])
In [36]: frame.sort_index()
Out[36]:
       d    a    b    c
one    4    5    6    7
three  0    1    2    3
In [37]: frame.sort_index(axis="columns")
Out[37]:
         a     b    c    d
three    1     2    3    0
one      5     6    7    4
```

2.6 Sorting and ranking

The data is sorted in ascending order by default but can be sorted in descending order, too:

```
In [38]: frame.sort_index(axis="columns",
   ascending=False)
Out[38]:
       d   c   b   a
three  0   3   2   1
one    4   7   6   5
```

To sort a Series by its values, use its `sort_values` method:

```
In [39]:s=pd.Series([4, 7, -3]);s.sort_values()
Out[39]:
2   -3
0    4
1    7
```

2.6 Sorting and ranking

Any missing values are sorted to the end of the Series by default, or its can be sorted to the start instead by using the `na_position` option:

```
In [40]: obj = pd.Series([4, np.nan, 7, np.nan,
    -3, 2])
In [41]: obj.sort_values()
Out[41]:
4    -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

2.6 Sorting and ranking

When sorting a DataFrame, you can use the data in one or more columns as the sort keys. To do so, pass one or more column names to `sort_values`:

```
In [42]: frame = pd.DataFrame({"b": [4, 7, -3,
    2], "a": [0, 1, 0, 1]})
In [43]: frame.sort_values(["a", "b"])
Out[43]:
    b   a
2  -3   0
0   4   0
3   2   1
1   7   1
```

2.6 Sorting and ranking

Ranking

Ranking assigns ranks from one through the number of valid data points in an array, starting from the lowest value. The rank methods for Series and DataFrame are the place to look; by default, rank breaks ties by assigning each group the mean rank:

```
In [44]: obj = pd.Series([7, -5, 7, 4, 2, 0])
In [44]: obj.rank()
Out[44]:
0    5.5
1    1.0
2    5.5
3    4.0
4    3.0
5    2.0
dtype: float64
```

2.6 Sorting and ranking

Ranks can also be assigned according to the order in which they're observed in the data by using the `method = "first"` argument:

```
In [45]: obj.rank(method="first")
Out[45]:
0    5.0
1    1.0
2    6.0
3    4.0
4    3.0
5    2.0
dtype: float64
```

You can also rank in descending order, using the `ascending = False` argument.

2.6 Sorting and ranking

DataFrame can compute ranks over the rows or the columns. By default, it ranks across the rows. Use `axis = 'columns'` instead to rank across the columns.

See Table 5-6 for a list of tie-breaking methods available.

Table 5-6. Tie-breaking methods with rank

Method	Description
`"average"`	Default: assign the average rank to each entry in the equal group
`"min"`	Use the minimum rank for the whole group
`"max"`	Use the maximum rank for the whole group
`"first"`	Assign ranks in the order the values appear in the data
`"dense"`	Like `method="min"`, but ranks always increase by 1 between groups rather than the number of equal elements in a group

2.7 Axis index with duplicated labels

Let's consider a small Series with duplicate indices:

```
In [46]: obj = pd.Series(np.arange(4),
    index=["a", "a", "b", "c"])
In [46]: obj
Out[46]:
a    0
a    1
b    2
c    3
dtype: int64
```

The `is_unique` property of the index can tell you whether or not its labels are unique:

```
In [47]: obj.index.is_unique
Out[47]: False
```

2.7 Axis index with duplicated labels

Data selection is one of the main things that behaves differently with duplicates. Indexing a label with multiple entries returns a Series, while single entries return a scalar value:

```
In [48]: obj["a"]
Out[48]:
a    0
a    1
dtype: int64
In [262]: obj["c"]
Out[262]: 4
```

This can make your code more complicated, as the output type from indexing can vary based on whether or not a label is repeated.

The same logic extends to indexing rows (or columns) in a DataFrame.

3. Summarizing and Computing Descriptive Statistics I

pandas objects are equipped with a set of common mathematical and statistical methods. Most of these fall into the category of reductions or summary statistics, methods that extract a single value (like the sum or mean) from a Series, or a Series of values from the rows or columns of a DataFrame. Compared with the similar methods found on NumPy arrays, they have built-in handling for missing data. Consider a small DataFrame:

```
In [49]: df = pd.DataFrame([[1.4, np.nan],
    [7.1, -4.5], [np.nan, np.nan], [0.75, -1.3]],
       index=["a", "b", "c", "d"],
       columns=["one", "two"])
In [50]: df
Out[50]:
```

3. Summarizing and Computing Descriptive Statistics II

```
     one     two
a   1.40     NaN
b   7.10    -4.5
c    NaN     NaN
d   0.75    -1.3
```

Calling DataFrame's sum method returns a Series containing column sums:

```
In [50]: df.sum()
Out[50]:
one    9.25
two   -5.80
dtype: float64
```

3. Summarizing and Computing Descriptive Statistics III

Passing axis="columns" or axis=1 sums across the columns instead:

```
In [51]: df.sum(axis="columns")
Out[51]:
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

By default, NaN value is skipped. We can disable this with the `skipna` option.

3. Summarizing and Computing Descriptive Statistics IV

```
In [52]: df.sum(skipna = False)
Out[52]:
one    NaN
two    NaN
dtype: float64
```

Some aggregations, like mean, require at least one non-NA value to yield a value result.

See Table 5-7 for a list of common options for each reduction method.

3. Summarizing and Computing Descriptive Statistics V

Table 5-7. Options for reduction methods

Method	Description
`axis`	Axis to reduce over; "index" for DataFrame's rows and "columns" for columns
`skipna`	Exclude missing values; `True` by default
`level`	Reduce grouped by level if the axis is hierarchically indexed (MultiIndex)

See Table 5-8 for a full list of summary statistics and related methods.

3. Summarizing and Computing Descriptive Statistics VI

Table 5-8. Descriptive and summary statistics

Method	Description
`count`	Number of non-NA values
`describe`	Compute set of summary statistics
`min, max`	Compute minimum and maximum values
`argmin, argmax`	Compute index locations (integers) at which minimum or maximum value is obtained, respectively; not available on DataFrame objects
`idxmin, idxmax`	Compute index labels at which minimum or maximum value is obtained, respectively
`quantile`	Compute sample quantile ranging from 0 to 1 (default: 0.5)
`sum`	Sum of values
`mean`	Mean of values
`median`	Arithmetic median (50% quantile) of values
`mad`	Mean absolute deviation from mean value

3. Summarizing and Computing Descriptive Statistics VII

`prod`	Product of all values
`var`	Sample variance of values
`std`	Sample standard deviation of values
`skew`	Sample skewness (third moment) of values
`kurt`	Sample kurtosis (fourth moment) of values
`cumsum`	Cumulative sum of values
`cummin, cummax`	Cumulative minimum or maximum of values, respectively
`cumprod`	Cumulative product of values
`diff`	Compute first arithmetic difference (useful for time series)
`pct_change`	Compute percent changes

3.1 Correlation and Covariance

Some summary statistics, like correlation and covariance, are computed from pairs of arguments.

When computing covariance (`cov()`) and correlation (`corr()`) between two Series, pandas will takes two steps:

- **Alignment**: pandas aligns the Series by their indexes, which means it will consider only the indexes that exist in both Series.
- **Handling NaN values**: The overlapping indexes that result in NaN after alignment will be automatically excluded from the calculations.

Computing covariance and correlation on a DataFrame is equivalent to computing covariance and correlation between all Series in it

3.1 Correlation and Covariance

```
In [52]: s1 = pd.Series([1, 2, 3, 4, 5],
   index=[0, 1, 2, 3, 4])
s2 = pd.Series([5, 4, 3, 2], index=[0, 1, 2, 3])
In [53]: s1.corr(s2)
Out[53]: -1.0
In [54]: data = {'A': [1, 2, 3, 4, 5], 'B': [5,
   4, 3, 2, 1], 'C': [2, 3, 4, 5, 6]}
df = pd.DataFrame(data)
In [55]: df.cov()
Out[55]:
     A    B    C
A  2.5 -2.5  2.5
B -2.5  2.5 -2.5
C  2.5 -2.5  2.5
```

3.1 Correlation and Covariance

Using DataFrame's `corrwith()` method, you can compute pair-wise correlations between a DataFrame's columns or rows with another Series or DataFrame. Passing a Series returns a Series with the correlation value computed for each column:

```
In [56]: df.corrwith(s1)
Out[56]:
A     1.0
B    -1.0
C     1.0
dtype: float64
```

3.2 Unique Values, Value Counts, and Membership

Another class of related methods extracts information about the values contained in a one-dimensional Series. To illustrate these, consider this example:

```
In [53]: obj = pd.Series(["c", "a", "d", "a",
    "a", "b", "b", "c", "c"])
```

The first function is unique, which gives you an array of the unique values in a Series:

```
In [54]: uniques = obj.unique()
In [55]: uniques
Out[55]: array(['c', 'a', 'd', 'b'],
    dtype=object)
```

3.2 Unique Values, Value Counts, and Membership

`value_counts` computes a Series containing value frequencies:

```
In [56]: obj.value_counts()
Out[56]:
c    3
a    3
b    2
d    1
dtype: int64
```

The Series is sorted by value in descending order as a convenience. `value_counts` is also available as a toplevel pandas method that can be used with NumPy arrays or other Python sequences.

3.2 Unique Values, Value Counts, and Membership

The `isin()` method performs a vectorized set membership check, meaning it efficiently checks whether each value in a Series or column of a DataFrame belongs to a specified set of values. This method is particularly useful for filtering a dataset down to a subset of values based on criteria specified by the user.

Related to `isin` is the `Index.get_indexer` method, which gives you an index array from an array of possibly nondistinct values into another array of distinct values.

3.2 Unique Values, Value Counts, and Membership

```
In [57]: mask = obj.isin(["b", "c"])
In [58]: mask
Out[58]:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
```

3.2 Unique Values, Value Counts, and Membership

```
In [59]: to_match = pd.Series(["c", "a", "b",
    "b", "c", "a"])
In [60]: unique_vals = pd.Series(["c", "b", "a"])
In [61]: indices =
    pd.Index(unique_vals).get_indexer(to_match)
In [62]: indices
Out[62]: array([0, 2, 1, 1, 0, 2])
```

3.2 Unique Values, Value Counts, and Membership

See Table 5-9 for a reference on these methods.

Table 5-9. Unique, value counts, and set membership methods

Method	Description
`isin`	Compute a Boolean array indicating whether each Series or DataFrame value is contained in the passed sequence of values
`get_indexer`	Compute integer indices for each value in an array into another array of distinct values; helpful for data alignment and join-type operations
`unique`	Compute an array of unique values in a Series, returned in the order observed
`value_counts`	Return a Series containing unique values as its index and frequencies as its values, ordered count in descending order

3.2 Unique Values, Value Counts, and Membership

There is also a `DataFrame.value_counts` method, but it computes counts considering each row of the DataFrame as a tuple to determine the number of occurrences of each distinct row.

```
In [308]: data = pd.DataFrame({"a": [1, 1, 1],
   "b": [0, 0, 1]})
In [309]: data
Out[309]:
   a  b
0  1  0
1  1  0
2  1  1
In [310]: data.value_counts()
Out[310]:
a  b
1  0  2
1  1  1
```

Conclusion

Conclusion

- This chapter provided a brief introduction to some basic pandas concepts. Understanding and mastering these fundamental concepts of pandas will greatly enhance your ability to handle, clean, transform, and analyze data efficiently.
- This foundation will pave the way for more advanced data analysis and machine learning tasks, making pandas an indispensable tool in the data scientist's toolkit.


Thank
you