SEAS 6414 – Python Applications in Data Analytics

THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

Dr. Adewale Akinfaderin

Lecture 4 - Spring 2024

Introduction to pandas

- Pandas is a pivotal tool for data cleaning and analysis in Python, offering fast and convenient data manipulation tools.
- It integrates well with numerical computing tools like NumPy and SciPy, analytical libraries like statsmodels and scikit-learn, and visualization libraries like matplotlib.
- Pandas extensively adopts NumPy's array-based computing style, particularly array-based functions and loop-less data processing.
- Unlike NumPy, which excels in handling homogeneously typed numerical array data, pandas is designed for tabular or heterogeneous data.
- Since its inception as an open source project in 2010, pandas has evolved into a comprehensive library for a wide range of real-world applications.

Introduction to pandas

- The development of pandas is backed by a community of over 2,500 contributors, addressing real-world data problems.
- The active and vibrant developer and user communities are a cornerstone of the success of pandas.
- Pandas' functionality and features have been refined over time through extensive community involvement and feedback.
- Its ability to work with different types of data makes pandas a versatile tool for data scientists and analysts.
- The library continues to grow and adapt, making it an essential tool in the modern data analysis toolkit.

Introduction to pandas Data Structures

To get started with pandas, you will need to get comfortable with its two workhorse data structures: *Series* and *DataFrame*. While they are not a universal solution for every problem, they provide a solid foundation for a wide variety of data tasks.

Series

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*. The simplest Series is formed from only an array of data:

```
In [14]: obj = pd.Series([4, 7, -5, 3])

In [15]: obj
Out[15]:
0    4
1    7
2   -5
3    3
dtype: int64
```

Introduction to pandas Data Structures

The string representation of a Series displayed interactively shows the index on the left and the values on the right. Since we did not specify an index for the data, a default one consisting of the integers `0` through `N - 1` (where `N` is the length of the data) is created. You can get the array representation and index object of the Series via its `array` and `index` attributes, respectively:

```
In [16]: obj.array
Out[16]:
<PandasArray>
[4, 7, -5, 3]
Length: 4, dtype: int64

In [17]: obj.index
Out[17]: RangeIndex(start=0, stop=4, step=1)
```

Introduction to pandas Data Structures

Often, you'll want to create a Series with an index identifying each data point with a label:

```
In [18]: obj2 = pd.Series([4, 7, -5, 3], index=["d", "b", "a", "c"])

In [19]: obj2
Out[19]:
d    4
b    7
a   -5
c    3
dtype: int64

In [20]: obj2.index
Out[20]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

Introduction to pandas Data Structures

Compared with NumPy arrays, you can use labels in the index when selecting single values or a set of values:

```
In [21]: obj2["a"]
Out[21]: -5

In [22]: obj2["d"] = 6

In [23]: obj2[["c", "a", "d"]]
Out[23]:
c    3
a   -5
d    6
dtype: int64
```

Here `["c", "a", "d"]` is interpreted as a list of indices, even though it contains strings instead of integers.

Introduction to pandas Data Structures

Using NumPy functions or NumPy-like operations, such as filtering with a Boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

```
In [24]: obj2[obj2 > 0]
Out[24]:
d    6
b    7
c    3
dtype: int64

In [25]: obj2 * 2
Out[25]:
d    12
b    14
a   -10
c     6
dtype: int64

In [26]: import numpy as np

In [27]: np.exp(obj2)
Out[27]:
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

Introduction to pandas Data Structures

Another way to think about a Series is as a fixed-length, ordered dictionary, as it is a mapping of index values to data values. It can be used in many contexts where you might use a dictionary:

```
In [28]: "b" in obj2
Out[28]: True

In [29]: "e" in obj2
Out[29]: False
```

Should you have data contained in a Python dictionary, you can create a Series from it by passing the dictionary:

```
In [30]: sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utah": 5000}

In [31]: obj3 = pd.Series(sdata)

In [32]: obj3
Out[32]:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

Introduction to pandas Data Structures

A Series can be converted back to a dictionary with its `to_dict` method:

```
In [33]: obj3.to_dict()
Out[33]: {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

When you are only passing a dictionary, the index in the resulting Series will respect the order of the keys according to the dictionary's `keys` method, which depends on the key insertion order. You can override this by passing an index with the dictionary keys in the order you want them to appear in the resulting Series:

```
In [34]: states = ["California", "Ohio", "Oregon", "Texas"]

In [35]: obj4 = pd.Series(sdata, index=states)

In [36]: obj4
Out[36]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

Introduction to pandas Data Structures

Here, three values found in `sdata` were placed in the appropriate locations, but since no value for `"California"` was found, it appears as `NaN` (Not a Number), which is considered in pandas to mark missing or *NA* values. Since `"Utah"` was not included in `states`, it is excluded from the resulting object.

I will use the terms "missing," "NA," or "null" interchangeably to refer to missing data. The `isna` and `notna` functions in pandas should be used to detect missing data:

```
In [37]: pd.isna(obj4)
Out[37]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool

In [38]: pd.notna(obj4)
Out[38]:
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

DataFrame

- A DataFrame represents a rectangular table of data, facilitating the organization and manipulation of potentially heterogeneous data.
- It contains an ordered, named collection of columns, where each column can hold different types of data such as numeric, string, Boolean, etc.
- The DataFrame possesses both a row and a column index, making it easy to reference specific rows or columns either by name or by numerical index.
- Conceptually, it can be thought of as a dictionary of Series, where each Series (column) shares the same index, thereby aligning the data across multiple types.

DataFrame

There are many ways to construct a DataFrame, though one of the most common is from a dictionary of equal-length lists or NumPy arrays:

```
data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
        "year": [2000, 2001, 2002, 2001, 2002, 2003],
        "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

The resulting DataFrame will have its index assigned automatically, as with Series, and the columns are placed according to the order of the keys in `data` (which depends on their insertion order in the dictionary):

```
In [50]: frame
Out[50]:
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

DataFrame

For large DataFrames, the `head` method selects only the first five rows:

```
In [51]: frame.head()
Out[51]:
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
```

Similarly, `tail` returns the last five rows:

```
In [52]: frame.tail()
Out[52]:
    state  year  pop
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

DataFrame

If you pass a column that isn't contained in the dictionary, it will appear with missing values in the result:

```
In [54]: frame2 = pd.DataFrame(data, columns=["year", "state", "pop", "debt"])

In [55]: frame2
Out[55]:
   year   state  pop debt
0  2000    Ohio  1.5  NaN
1  2001    Ohio  1.7  NaN
2  2002    Ohio  3.6  NaN
3  2001  Nevada  2.4  NaN
4  2002  Nevada  2.9  NaN
5  2003  Nevada  3.2  NaN

In [56]: frame2.columns
Out[56]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

DataFrame

A column in a DataFrame can be retrieved as a Series either by dictionary-like notation or by using the dot attribute notation:

```
In [57]: frame2["state"]
Out[57]:
0      Ohio
1      Ohio
2      Ohio
3    Nevada
4    Nevada
5    Nevada
Name: state, dtype: object

In [58]: frame2.year
Out[58]:
0    2000
1    2001
2    2002
3    2001
4    2002
5    2003
Name: year, dtype: int64
```

DataFrame

Rows can also be retrieved by position or name with the special `iloc` and `loc` attributes (more on this later in Selection on DataFrame with loc and iloc):

```
In [59]: frame2.loc[1]
Out[59]:
year     2001
state    Ohio
pop       1.7
debt      NaN
Name: 1, dtype: object

In [60]: frame2.iloc[2]
Out[60]:
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: 2, dtype: object
```

DataFrame

Columns can be modified by assignment. For example, the empty `debt` column could be assigned a scalar value or an array of values:

```
In [61]: frame2["debt"] = 16.5

In [62]: frame2
Out[62]:
   year   state  pop  debt
0  2000    Ohio  1.5  16.5
1  2001    Ohio  1.7  16.5
2  2002    Ohio  3.6  16.5
3  2001  Nevada  2.4  16.5
4  2002  Nevada  2.9  16.5
5  2003  Nevada  3.2  16.5

In [63]: frame2["debt"] = np.arange(6.)

In [64]: frame2
Out[64]:
   year   state  pop  debt
0  2000    Ohio  1.5   0.0
1  2001    Ohio  1.7   1.0
2  2002    Ohio  3.6   2.0
3  2001  Nevada  2.4   3.0
4  2002  Nevada  2.9   4.0
5  2003  Nevada  3.2   5.0
```

DataFrame

When you are assigning lists or arrays to a column, the value's length must match the length of the DataFrame. If you assign a Series, its labels will be realigned exactly to the DataFrame's index, inserting missing values in any index values not present:

```
In [65]: val = pd.Series([-1.2, -1.5, -1.7], index=[2, 4, 5])

In [66]: frame2["debt"] = val

In [67]: frame2
Out[67]:
   year   state  pop  debt
0  2000    Ohio  1.5   NaN
1  2001    Ohio  1.7   NaN
2  2002    Ohio  3.6  -1.2
3  2001  Nevada  2.4   NaN
4  2002  Nevada  2.9  -1.5
5  2003  Nevada  3.2  -1.7
```

Assigning a column that doesn't exist will create a new column.

DataFrame

Assigning a column that doesn't exist will create a new column.

The `del` keyword will delete columns like with a dictionary. As an example, I first add a new column of Boolean values where the `state` column equals `"Ohio"`:

```
In [68]: frame2["eastern"] = frame2["state"] == "Ohio"

In [69]: frame2
Out[69]:
   year   state  pop  debt  eastern
0  2000    Ohio  1.5   NaN     True
1  2001    Ohio  1.7   NaN     True
2  2002    Ohio  3.6  -1.2     True
3  2001  Nevada  2.4   NaN    False
4  2002  Nevada  2.9  -1.5    False
5  2003  Nevada  3.2  -1.7    False
```

DataFrame

Caution

New columns cannot be created with the `frame2.eastern` dot attribute notation.

The `del` method can then be used to remove this column:

```
In [70]: del frame2["eastern"]

In [71]: frame2.columns
Out[71]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Caution

The column returned from indexing a DataFrame is a *view* on the underlying data, not a copy. Thus, any in-place modifications to the Series will be reflected in the DataFrame. The column can be explicitly copied with the Series's `copy` method.

DataFrame

Another common form of data is a nested dictionary of dictionaries:

```
In [72]: populations = {"Ohio": {2000: 1.5, 2001: 1.7, 2002: 3.6},
   ....:                "Nevada": {2001: 2.4, 2002: 2.9}}
```

If the nested dictionary is passed to the DataFrame, pandas will interpret the outer dictionary keys as the columns, and the inner keys as the row indices:

```
In [73]: frame3 = pd.DataFrame(populations)

In [74]: frame3
Out[74]:
      Ohio  Nevada
2000   1.5     NaN
2001   1.7     2.4
2002   3.6     2.9
```

DataFrame

You can transpose the DataFrame (swap rows and columns) with similar syntax to a NumPy array:

```
In [75]: frame3.T
Out[75]:
        2000  2001  2002
Ohio     1.5   1.7   3.6
Nevada   NaN   2.4   2.9
```

> ⚠ **Warning**
>
> Note that transposing discards the column data types if the columns do not all have the same data type, so transposing and then transposing back may lose the previous type information. The columns become arrays of pure Python objects in this case.

The keys in the inner dictionaries are combined to form the index in the result. This isn't true if an explicit index is specified:

```
In [76]: pd.DataFrame(populations, index=[2001, 2002, 2003])
Out[76]:
      Ohio  Nevada
2001   1.7     2.4
2002   3.6     2.9
2003   NaN     NaN
```

Possible data inputs to the DataFrame constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
Dictionary of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame; all sequences must be the same length
NumPy structured/record array	Treated as the "dictionary of arrays" case
Dictionary of Series	Each value becomes a column; indexes from each Series are unioned together to form the result's row index if no explicit index is passed
Dictionary of dictionaries	Each inner dictionary becomes a column; keys are unioned to form the row index as in the "dictionary of Series" case
List of dictionaries or Series	Each item becomes a row in the DataFrame; unions of dictionary keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values are missing in the DataFrame result

DataFrame

If a DataFrame's `index` and `columns` have their `name` attributes set, these will also be displayed:

```
In [79]: frame3.index.name = "year"

In [80]: frame3.columns.name = "state"

In [81]: frame3
Out[81]:
state  Ohio  Nevada
year
2000    1.5     NaN
2001    1.7     2.4
2002    3.6     2.9
```

Unlike Series, DataFrame does not have a `name` attribute. DataFrame's `to_numpy` method returns the data contained in the DataFrame as a two-dimensional ndarray:

```
In [82]: frame3.to_numpy()
Out[82]:
array([[1.5, nan],
       [1.7, 2.4],
       [3.6, 2.9]])
```

Index Objects

pandas's Index objects are responsible for holding the axis labels (including a DataFrame's column names) and other metadata (like the axis name or names). Any array or other sequence of labels you use when constructing a Series or DataFrame is internally converted to an Index:

```
In [84]: obj = pd.Series(np.arange(3), index=["a", "b", "c"])

In [85]: index = obj.index

In [86]: index
Out[86]: Index(['a', 'b', 'c'], dtype='object')

In [87]: index[1:]
Out[87]: Index(['b', 'c'], dtype='object')
```

Index objects are immutable and thus can't be modified by the user:

```
index[1] = "d"  # TypeError
```

Index Objects

Immutability makes it safer to share Index objects among data structures:

```
In [88]: labels = pd.Index(np.arange(3))

In [89]: labels
Out[89]: Index([0, 1, 2], dtype='int64')

In [90]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)

In [91]: obj2
Out[91]:
0    1.5
1   -2.5
2    0.0
dtype: float64

In [92]: obj2.index is labels
Out[92]: True
```

Index Objects

In addition to being array-like, an Index also behaves like a fixed-size set:

```
In [93]: frame3
Out[93]:
state  Ohio  Nevada
year
2000    1.5     NaN
2001    1.7     2.4
2002    3.6     2.9

In [94]: frame3.columns
Out[94]: Index(['Ohio', 'Nevada'], dtype='object', name='state')

In [95]: "Ohio" in frame3.columns
Out[95]: True

In [96]: 2003 in frame3.index
Out[96]: False
```

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. Consider an example:

```
In [98]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=["d", "b", "a", "c"])

In [99]: obj
Out[99]:
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

Functionalities

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

```
In [100]: obj2 = obj.reindex(["a", "b", "c", "d", "e"])

In [101]: obj2
Out[101]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

Functionalities

For ordered data like time series, you may want to do some interpolation or filling of values when reindexing. The `method` option allows us to do this, using a method such as `ffill`, which forward-fills the values:

```
In [102]: obj3 = pd.Series(["blue", "purple", "yellow"], index=[0, 2, 4])

In [103]: obj3
Out[103]:
0      blue
2    purple
4    yellow
dtype: object

In [104]: obj3.reindex(np.arange(6), method="ffill")
Out[104]:
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

Functionalities

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

```
In [105]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
   .....:                      index=["a", "c", "d"],
   .....:                      columns=["Ohio", "Texas", "California"])

In [106]: frame
Out[106]:
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8

In [107]: frame2 = frame.reindex(index=["a", "b", "c", "d"])

In [108]: frame2
Out[108]:
   Ohio  Texas  California
a   0.0    1.0         2.0
b   NaN    NaN         NaN
c   3.0    4.0         5.0
d   6.0    7.0         8.0
```

Functionalities

The columns can be reindexed with the `columns` keyword:

```
In [109]: states = ["Texas", "Utah", "California"]

In [110]: frame.reindex(columns=states)
Out[110]:
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

Because `"Ohio"` was not in `states`, the data for that column is dropped from the result.

Another way to reindex a particular axis is to pass the new axis labels as a positional argument and then specify the axis to reindex with the `axis` keyword:

```
In [111]: frame.reindex(states, axis="columns")
Out[111]:
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

Functionalities

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.

Functionalities

`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.

Dropping Entries from an Axis

Dropping one or more entries from an axis is simple if you already have an index array or list without those entries, since you can use the `reindex` method or `.loc`-based indexing. As that can require a bit of munging and set logic, the `drop` method will return a new object with the indicated value or values deleted from an axis:

```
In [113]: obj = pd.Series(np.arange(5.), index=["a", "b", "c", "d", "e"])

In [114]: obj
Out[114]:
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

Functionalities

```
In [115]: new_obj = obj.drop("c")

In [116]: new_obj
Out[116]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64

In [117]: obj.drop(["d", "c"])
Out[117]:
a    0.0
b    1.0
e    4.0
dtype: float64
```

Functionalities

With DataFrame, index values can be deleted from either axis. To illustrate this, we first create an example DataFrame:

```
In [118]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=["Ohio", "Colorado", "Utah", "New York"],
   .....:                     columns=["one", "two", "three", "four"])

In [119]: data
Out[119]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Functionalities

Calling `drop` with a sequence of labels will drop values from the row labels (axis 0):

```
In [120]: data.drop(index=["Colorado", "Ohio"])
Out[120]:
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

To drop labels from the columns, instead use the `columns` keyword:

```
In [121]: data.drop(columns=["two"])
Out[121]:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15
```

Functionalities

You can also drop values from the columns by passing `axis=1` (which is like NumPy) or `axis="columns"`:

```
In [122]: data.drop("two", axis=1)
Out[122]:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15

In [123]: data.drop(["two", "four"], axis="columns")
Out[123]:
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

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. Here are some examples of this:

```
In [124]: obj = pd.Series(np.arange(4.), index=["a", "b", "c", "d"])

In [125]: obj
Out[125]:
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64

In [126]: obj["b"]
Out[126]: 1.0

In [127]: obj[1]
Out[127]: 1.0

In [128]: obj[2:4]
Out[128]:
c    2.0
d    3.0
dtype: float64
```

Indexing, Selection, and Filtering

```
In [129]: obj[["b", "a", "d"]]
Out[129]:
b    1.0
a    0.0
d    3.0
dtype: float64

In [130]: obj[[1, 3]]
Out[130]:
b    1.0
d    3.0
dtype: float64

In [131]: obj[obj < 2]
Out[131]:
a    0.0
b    1.0
dtype: float64
```

Indexing, Selection, and Filtering

While you can select data by label this way, the preferred way to select index values is with the special `loc` operator:

```
In [132]: obj.loc[["b", "a", "d"]]
Out[132]:
b    1.0
a    0.0
d    3.0
dtype: float64
```

The reason to prefer `loc` is because of the different treatment of integers when indexing with `[]`. 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. For example:

```
In [133]: obj1 = pd.Series([1, 2, 3], index=[2, 0, 1])

In [134]: obj2 = pd.Series([1, 2, 3], index=["a", "b", "c"])

In [135]: obj1
Out[135]:
2    1
0    2
1    3
dtype: int64
```

Indexing, Selection, and Filtering

```
In [136]: obj2
Out[136]:
a    1
b    2
c    3
dtype: int64

In [137]: obj1[[0, 1, 2]]
Out[137]:
0    2
1    3
2    1
dtype: int64

In [138]: obj2[[0, 1, 2]]
Out[138]:
a    1
b    2
c    3
dtype: int64
```

Indexing, Selection, and Filtering

Since `loc` operator indexes exclusively with labels, there is also an `iloc` operator that indexes exclusively with integers to work consistently whether or not the index contains integers:

```
In [139]: obj1.iloc[[0, 1, 2]]
Out[139]:
2    1
0    2
1    3
dtype: int64

In [140]: obj2.iloc[[0, 1, 2]]
Out[140]:
a    1
b    2
c    3
dtype: int64
```

Indexing, Selection, and Filtering

Caution

You can also slice with labels, but it works differently from normal Python slicing in that the endpoint is inclusive:

```
In [141]: obj2.loc["b":"c"]
Out[141]:
b    2
c    3
dtype: int64
```

Assigning values using these methods modifies the corresponding section of the Series:

```
In [142]: obj2.loc["b":"c"] = 5

In [143]: obj2
Out[143]:
a    1
b    5
c    5
dtype: int64
```

Indexing, Selection, and Filtering

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

```
In [144]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=["Ohio", "Colorado", "Utah", "New York"],
   .....:                     columns=["one", "two", "three", "four"])

In [145]: data
Out[145]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Indexing, Selection, and Filtering

```
In [146]: data["two"]
Out[146]:
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64

In [147]: data[["three", "one"]]
Out[147]:
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
```

Indexing, Selection, and Filtering

Indexing like this has a few special cases. The first is slicing or selecting data with a Boolean array:

```
In [148]: data[:2]
Out[148]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7

In [149]: data[data["three"] > 5]
Out[149]:
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Indexing, Selection, and Filtering

The row selection syntax `data[:2]` is provided as a convenience. Passing a single element or a list to the `[]` operator selects columns.

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 [150]: data < 5
Out[150]:
            one    two  three   four
Ohio       True   True   True   True
Colorado   True  False  False  False
Utah      False  False  False  False
New York  False  False  False  False
```

Indexing, Selection, and Filtering

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

```
In [151]: data[data < 5] = 0

In [152]: data
Out[152]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Selection on DataFrame with loc and iloc

Like Series, DataFrame has special attributes `loc` and `iloc` for label-based and integer-based indexing, respectively. Since DataFrame is two-dimensional, you can select a subset of the rows and columns with NumPy-like notation using either axis labels (`loc`) or integers (`iloc`).

Indexing, Selection, and Filtering

As a first example, let's select a single row by label:

```
In [153]: data
Out[153]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15

In [154]: data.loc["Colorado"]
Out[154]:
one      0
two      5
three    6
four     7
Name: Colorado, dtype: int64
```

Indexing, Selection, and Filtering

The result of selecting a single row is a Series with an index that contains the DataFrame's column labels. To select multiple roles, creating a new DataFrame, pass a sequence of labels:

```
In [155]: data.loc[["Colorado", "New York"]]
Out[155]:
          one  two  three  four
Colorado    0    5      6     7
New York   12   13     14    15
```

You can combine both row and column selection in `loc` by separating the selections with a comma:

```
In [156]: data.loc["Colorado", ["two", "three"]]
Out[156]:
two      5
three    6
Name: Colorado, dtype: int64
```

Indexing, Selection, and Filtering

We'll then perform some similar selections with integers using `iloc`:

```
In [157]: data.iloc[2]
Out[157]:
one       8
two       9
three    10
four     11
Name: Utah, dtype: int64

In [158]: data.iloc[[2, 1]]
Out[158]:
          one  two  three  four
Utah        8    9     10    11
Colorado    0    5      6     7
```

Indexing, Selection, and Filtering

```
In [159]: data.iloc[2, [3, 0, 1]]
Out[159]:
four    11
one      8
two      9
Name: Utah, dtype: int64

In [160]: data.iloc[[1, 2], [3, 0, 1]]
Out[160]:
          four  one  two
Colorado     7    0    5
Utah        11    8    9
```

Indexing, Selection, and Filtering

Both indexing functions work with slices in addition to single labels or lists of labels:

```
In [161]: data.loc[:"Utah", "two"]
Out[161]:
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int64

In [162]: data.iloc[:, :3][data.three > 5]
Out[162]:
          one  two  three
Colorado    0    5      6
Utah        8    9     10
New York   12   13     14
```

Indexing, Selection, and Filtering

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

Indexing, Selection, and Filtering

`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

Function Application and Mapping

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

```
In [223]: frame = pd.DataFrame(np.random.standard_normal((4, 3)),
   .....:                      columns=list("bde"),
   .....:                      index=["Utah", "Ohio", "Texas", "Oregon"])

In [224]: frame
Out[224]:
               b         d         e
Utah   -0.204708  0.478943 -0.519439
Ohio   -0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189 -1.296221

In [225]: np.abs(frame)
Out[225]:
               b         d         e
Utah    0.204708  0.478943  0.519439
Ohio    0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189  1.296221
```

Function Application and Mapping

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

```
In [226]: def f1(x):
   .....:     return x.max() - x.min()

In [227]: frame.apply(f1)
Out[227]:
b    1.802165
d    1.684034
e    2.689627
dtype: float64
```

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.

Function Application and Mapping

If you pass `axis="columns"` to `apply`, the function will be invoked once per row instead. A helpful way to think about this is as "apply across the columns":

```
In [228]: frame.apply(f1, axis="columns")
Out[228]:
Utah      0.998382
Ohio      2.521511
Texas     0.676115
Oregon    2.542656
dtype: float64
```

Many of the most common array statistics (like `sum` and `mean`) are DataFrame methods, so using `apply` is not necessary.

Function Application and Mapping

The function passed to `apply` need not return a scalar value; it can also return a Series with multiple values:

```
In [229]: def f2(x):
   .....:     return pd.Series([x.min(), x.max()], index=["min", "max"])

In [230]: frame.apply(f2)
Out[230]:
            b         d         e
min -0.555730  0.281746 -1.296221
max  1.246435  1.965781  1.393406
```

Function Application and Mapping

Element-wise Python functions can be used, too. Suppose you wanted to compute a formatted string from each floating-point value in `frame`. You can do this with `applymap`:

```
In [231]: def my_format(x):
   .....:     return f"{x:.2f}"

In [232]: frame.applymap(my_format)
Out[232]:
            b     d      e
Utah    -0.20  0.48  -0.52
Ohio    -0.56  1.97   1.39
Texas    0.09  0.28   0.77
Oregon   1.25  1.01  -1.30
```

Function Application and Mapping

The reason for the name `applymap` is that Series has a `map` method for applying an element-wise function:

```
In [233]: frame["e"].map(my_format)
Out[233]:
Utah      -0.52
Ohio       1.39
Texas      0.77
Oregon    -1.30
Name: e, dtype: object
```

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 [234]: obj = pd.Series(np.arange(4), index=["d", "a", "b", "c"])

In [235]: obj
Out[235]:
d    0
a    1
b    2
c    3
dtype: int64

In [236]: obj.sort_index()
Out[236]:
a    1
b    2
c    3
d    0
dtype: int64
```

Sorting and Ranking

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

```
In [237]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
   .....:                      index=["three", "one"],
   .....:                      columns=["d", "a", "b", "c"])

In [238]: frame
Out[238]:
       d  a  b  c
three  0  1  2  3
one    4  5  6  7

In [239]: frame.sort_index()
Out[239]:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3

In [240]: frame.sort_index(axis="columns")
Out[240]:
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

Sorting and Ranking

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

```
In [241]: frame.sort_index(axis="columns", ascending=False)
Out[241]:
       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 [242]: obj = pd.Series([4, 7, -3, 2])

In [243]: obj.sort_values()
Out[243]:
2   -3
3    2
0    4
1    7
dtype: int64
```

Sorting and 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 [251]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])

In [252]: obj.rank()
Out[252]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

Summarizing and Computing Descriptive Statistics

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 [267]: 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 [268]: df
Out[268]:
    one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3
```

Summarizing and Computing Descriptive Statistics

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

```
In [269]: df.sum()
Out[269]:
one    9.25
two   -5.80
dtype: float64
```

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

```
In [270]: df.sum(axis="columns")
Out[270]:
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

Summarizing and Computing Descriptive Statistics

When an entire row or column contains all NA values, the sum is 0, whereas if any value is not NA, then the result is NA. This can be disabled with the `skipna` option, in which case any NA value in a row or column names the corresponding result NA:

```
In [271]: df.sum(axis="index", skipna=False)
Out[271]:
one   NaN
two   NaN
dtype: float64

In [272]: df.sum(axis="columns", skipna=False)
Out[272]:
a     NaN
b    2.60
c     NaN
d   -0.55
dtype: float64
```

Summarizing and Computing Descriptive Statistics

Some methods are neither reductions nor accumulations. `describe` is one such example, producing multiple summary statistics in one shot:

```
In [276]: df.describe()
Out[276]:
             one       two
count   3.000000  2.000000
mean    3.083333 -2.900000
std     3.493685  2.262742
min     0.750000 -4.500000
25%     1.075000 -3.700000
50%     1.400000 -2.900000
75%     4.250000 -2.100000
max     7.100000 -1.300000
```

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 [289]: 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 [290]: uniques = obj.unique()

In [291]: uniques
Out[291]: array(['c', 'a', 'd', 'b'], dtype=object)
```

Unique Values, Value Counts, and Membership

The unique values are not necessarily returned in the order in which they first appear, and not in sorted order, but they could be sorted after the fact if needed (`uniques.sort()`). Relatedly, `value_counts` computes a Series containing value frequencies:

```
In [292]: obj.value_counts()
Out[292]:
c    3
a    3
b    2
d    1
Name: count, dtype: int64
```

Unique Values, Value Counts, and Membership

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

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 [307]: data = pd.DataFrame({"a": [1, 1, 1, 2, 2], "b": [0, 0, 1, 0, 0]})

In [308]: data
Out[308]:
   a  b
0  1  0
1  1  0
2  1  1
3  2  0
4  2  0

In [309]: data.value_counts()
Out[309]:
a  b
1  0    2
2  0    2
1  1    1
Name: count, dtype: int64
```

SEAS 6414 – Python Applications in Data Analytics

THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

Dr. Adewale Akinfaderin

Lecture 4 - Spring 2024