Intro. Comp. for Data Science (FMI08)

Dr. Nono Saha

May 26, 2023

Max Planck Institute for Mathematics in the Sciences
University of Leipzig/ScaDS.AI

Spring 2023

Course plan

1. Introduction to **pandas**
2. Series
3. DataFrame
4. Missing value problems and native Nas

Introduction to pandas

pandas?

pandas implements data frames in Python - it takes much of its inspiration from R and NumPy.

pandas aims to be the fundamental high-level building block for practical, real-world data analysis in Python. Additionally, it seeks to become the most powerful and flexible open-source data analysis/manipulation tool available in any language.

Key features:

- DataFrame object class
- Reading and writing tabular data
- Data munging (filtering, grouping, summarizing, joining, etc.)
- Data reshaping

pandas - Series

Series - create a pandas series from a list

The columns of a DataFrame are constructed as Series - a 1d array-like object containing values of the same type (similar to an ndarray).

```
import pandas as pd
```

```
pd.Series([1,2,3,4])
## 0    1
## 1    2
## 2    3
## 3    4
## dtype: int64


pd.Series(["C","B","A"])
## 0    C
## 1    B
## 2    A
## dtype: object
```

```
pd.Series(range(5))
## 0    0
## 1    1
## 2    2
## 3    3
## 4    4
## dtype: int64

pd.Series([1,"A",True])
## 0       1
## 1       A
## 2    True
## dtype: object
```

pandas - Series: basic methods

Once constructed the components of a series can be accessed via `array()` and `index()` methods.

```
s = pd.Series([4,2,1,3])
s.array
## <PandasArray>
## [4, 2, 1, 3]
## Length: 4, dtype: int64

s.index
## RangeIndex(start=0, stop=4, step=1)
```

An index can also be explicitly provided when constructing a Series,

```
t = pd.Series([4,2,1,3], index=["a","b","c","d"])
## a    4
## b    2
## c    1
## d    3
## dtype: int64
```

```
t.array
## <PandasArray>
## [4, 2, 1, 3]
## Length: 4, dtype: int64

t.index
```

pandas - Series + NumPy

Series objects are compatible with NumPy like functions (vectorized)

```
t = pd.Series([4,2,1,3], index=["a","b","c","d"])
```

```
t+1
## a    5
## b    3
## c    2
## d    4
## dtype: int64

np.log(t)
## a    1.386294
## b    0.693147
## c    0.000000
## d    1.098612
## dtype: float64
```

```
np.exp(-t**2/2)
## a    0.000335
## b    0.135335
## c    0.606531
## d    0.011109
## dtype: float64
```

pandas - Series indexing

`Series` can be indexed in the same way as `NumPy` arrays with the addition of being able to use label(s) when selecting elements.

```
t = pd.Series([4,2,1,3], index=["a","b","c","d"])
```

```
t[1]
## 2

t[[1,2]]
## b    2
## c    1
## dtype: int64

t["c"]
## 1

t[["a","d"]]
## a    4
## d    3
## dtype: int64
```

```
t[t == 3]
## d    3
## dtype: int64

t[t % 2 == 0]
## a    4
## b    2
## dtype: int64

t["d"] = 6
t
## a    4
## b    2
## c    1
## d    6
```

pandas - Series: index alignment

When performing (arithmetic) operations on Series, they will attempt to align by their index. Let us consider the following `Series`:

```
m = pd.Series([1,2,3,4], index = ["a","b","c","d"])
n = pd.Series([4,3,2,1], index = ["d","c","b","a"])
o = pd.Series([1,1,1,1,1], index = ["b","d","a","c","e"])
```

Questions

For each of the following operations, what is the output?

- m + n
- n + m
- n + 0
- m + 0

pandas - Series and dicts

We can construct Series from dicts, in which case the keys are used to form the index,

```
d = {"anna": "A+", "bob": "B-", "carol": "C", "dave": "D+"}
pd.Series(d)
## anna     A+
## bob      B-
## carol     C
## dave     D+
## dtype: object
```

Index order will follow key order unless overridden by index,

```
pd.Series(d, index = ["dave","carol","bob","anna"])

## dave     D+
## carol     C
## bob      B-
## anna     A+
## dtype: object
```

pandas - Series: string series

Series containing strings can be accessed via the `str` attribute,

```
s = pd.Series(["the quick", "brown fox", "jumps over", "a lazy dog"])
s
## 0     the quick
## 1     brown fox
## 2    jumps over
## 3    a lazy dog
## dtype: object

s.str.upper()
## 0     THE QUICK
## 1     BROWN FOX
## 2    JUMPS OVER
## 3    A LAZY DOG
## dtype: object
```

```
s.str.split(" ").str[1]
## 0    quick
## 1      fox
## 2     over
## 3     lazy
## dtype: object

pd.Series([1,2,3]).str
## AttributeError: Can only use .str accessor with string
```

pandas - Series: categorical Series

```
pd.Series(["Mon", "Tue", "Wed", "Thur", "Fri"])
## 0      Mon
## 1      Tue
## 2      Wed
## 3     Thur
## 4      Fri
## dtype: object
```

```
pd.Series(["Mon", "Tue", "Wed", "Thur", "Fri"], dtype="category")
## 0      Mon
## 1      Tue
## 2      Wed
## 3     Thur
## 4      Fri
## dtype: category
## Categories (5, object): ['Fri', 'Mon', 'Thur', 'Tue', 'Wed']
```

```
pd.Series(["Mon", "Tue", "Wed", "Thur", "Fri"], dtype=pd.CategoricalDtype(ordered=True))
## 0      Mon
## 1      Tue
## 2      Wed
## 3     Thur
## 4      Fri
## dtype: category
## Categories (5, object): ['Fri' < 'Mon' < 'Thur' < 'Tue' < 'Wed']
```

pandas - DataFrame

pandas - DataFrame: constructing DataFrames

`panda` data frames can also be constructed via `DataFrame()`, general this is done via dict of columns:

```
n = 5
d = {
"id":     np.random.randint(100, 999, n),
"weight": np.random.normal(70, 20, n),
"height": np.random.normal(170, 15, n),
"date":   pd.date_range(start='2/1/2022', periods=n, freq='D')
}
df = pd.DataFrame(d)
df

##     id      weight      height       date
## 0  168  102.915535  188.677769 2022-02-01
## 1  615   71.767364  155.907801 2022-02-02
## 2  346   76.666059  171.386839 2022-02-03
## 3  390   74.735465  173.151008 2022-02-04
## 4  556   50.538488  183.083407 2022-02-05
```

pandas - DataFrame: from nparray

For 2d ndarrays, it is also possible to construct a DataFrame - generally, providing column names and row names (indexes) is a good idea.

```
pd.DataFrame(
np.diag([1,2,3]),
columns = ["x","y","z"]
)
##    x  y  z
## 0  1  0  0
## 1  0  2  0
## 2  0  0  3

pd.DataFrame(
np.diag([1,2,3]),
columns = ["x","y","z"]
)
##    x  y  z
## 0  1  0  0
## 1  0  2  0
## 2  0  0  3
```

```
pd.DataFrame(
np.tri(5,3,-1),
columns = ["x","y","z"],
index = ["a","b","c","d","e"]
)
##        x    y    z
## a  0.0  0.0  0.0
## b  1.0  0.0  0.0
## c  1.0  1.0  0.0
## d  1.0  1.0  1.0
## e  1.0  1.0  1.0
```

pandas - DataFrame: indexing

```
df[0]
## KeyError: 0

df["id"]
## 0    168
## 1    615
## 2    346
## 3    390
## 4    556
## Name: id, dtype: int64

df.id
## 0    168
## 1    615
## 2    346
## 3    390
## 4    556
## Name: id, dtype: int64
```

```
df[1:3]
##      id     weight      height
     date
## 1  615  71.767364  155.907801
    2022-02-02
## 2  346  76.666059  171.386839
    2022-02-03

df[0::2]
##      id      weight      height
      date
## 0  168  102.915535  188.677769
    2022-02-01
## 2  346   76.666059  171.386839
    2022-02-03
## 4  556   50.538488  183.083407
    2022-02-05
```

pandas - DataFrame: indexing by position

```
df.iloc[1] && df.iloc[[1]]
## What is the difference between the two instructions?

df.iloc[0:2]
## id  weight  height  date
## 0  168  102.915535  188.677769  2022-02-01
## 1  615   71.767364  155.907801  2022-02-02

df.iloc[lambda x: x.index % 2 != 0]
##      id     weight      height      date
## 1  615  71.767364  155.907801  2022-02-02
## 3  390  74.735465  173.151008  2022-02-04
```

```
df.iloc[1:3,1:3]
##         weight      height
## 1  71.767364  155.907801
## 2  76.666059  171.386839

df.iloc[0:3, [0,3]]
##      id        date
## 0  168 2022-02-01
## 1  615 2022-02-02
## 2  346 2022-02-03

df.iloc[0:3, [True, True, False, False]]
##      id       weight
## 0  168  102.915535
## 1  615   71.767364
## 2  346   76.666059
```

pandas - DataFrame: indexing by name

```
df.loc[["anna"]]
## id weight height date
## anna  168  102.915535
    188.677769 2022-02-01

df.loc["bob":"dave"]
## id weight height date
## bob 615 71.76 155.9 2022-02-02
## carol 346 76.6 171.3 2022-02-03
## dave 390 74.73 173.15 2022-02-04

df.loc[df.id < 300]
## id weight height date
## anna 168 102.91 188.67
    2022-02-01
```

```
df.loc[:, "date"]
## anna    2022-02-01
## bob     2022-02-02
## carol   2022-02-03
## dave    2022-02-04
## erin    2022-02-05
## Name: date, dtype:
    datetime64[ns]

df.loc[["bob","erin"], "
    weight":"height"]
## weight height
## bob   71.76  155.9
## erin  50.53  183.08

df.loc[0:2, "weight":"height"
    ]
## ???
```

pandas - DataFrame: views and copies

In general, most pandas operations will generate a new object, but some will return views, mostly the later occurring with subsetting.

```
d = pd.DataFrame(np.arange(6).reshape(3,2), columns = ["x","y"])
```

```
v = d.iloc[0:2,0:2]
v
##    x  y
## 0  0  1
## 1  2  3

d.iloc[0,1] = -1
v
##    x  y
## 0  0 -1
## 1  2  3
```

```
v.iloc[0,0] = np.pi
v
##           x  y
## 0  3.141593 -1
## 1  2.000000  3

d
##    x  y
## 0  0 -1
## 1  2  3
## 2  4  5
```

pandas - DataFrame: filtering rows

The `query()` method can be used for filtering rows. It evaluates a string expression in the context of the data frame.

```
df = pd.DataFrame(d)
df.query('date == "2022-02-01"')
##        id      weight      height       date
## anna  168  102.915535  188.677769 2022-02-01

df.query('weight > 50')
##         id      weight      height       date
## anna   168  102.915535  188.677769 2022-02-01
## bob    615   71.767364  155.907801 2022-02-02
## carol  346   76.666059  171.386839 2022-02-03
## dave   390   74.735465  173.151008 2022-02-04

df.query('weight > 50 & height < 165')
##       id     weight      height       date
## bob  615  71.767364  155.907801 2022-02-02

qid = 414
df.query('id == @qid')
```

pandas - DataFrame: element access

```
df
##         id      weight      height       date
## anna   168  102.915535  188.677769 2022-02-01
## bob    615   71.767364  155.907801 2022-02-02
## carol  346   76.666059  171.386839 2022-02-03
## dave   390   74.735465  173.151008 2022-02-04
## erin   556   50.538488  183.083407 2022-02-05
```

```
df[0,0]
## KeyError: (0, 0)

df.iat[0,0]
## 168

df.id[0]
## 168

df[0:1].id[0]
## 168
```

```
df["anna", "id"]
## KeyError: ('anna', 'id')

df.at["anna", "id"]
## 168

df["id"]["anna"]
## 168

df["id"][0]
## 168
```

pandas - DataFrame: properties of a DataFrame

```
df
##          id      weight      height       date
## anna   168  102.915535  188.677769 2022-02-01
## bob    615   71.767364  155.907801 2022-02-02
## carol  346   76.666059  171.386839 2022-02-03
## dave   390   74.735465  173.151008 2022-02-04
## erin   556   50.538488  183.083407 2022-02-05
```

```
df.size
## 20

df.shape
## (5, 4)

df.info()
## Try it on your computer and analyze the output.
```

```
df.dtypes
## id                 int64
## weight           float64
## height           float64
## date      datetime64[ns]
## dtype: object

df.describe()
```

pandas - DataFrame: selecting columns

Beyond the use of `loc()` and `iloc()`, there is also the `filter()` method which can
be used to select columns (or indices) by name with pattern-matching.

```
df.filter(items=["id","weight"])
##         id      weight
## anna   168  102.915535
## bob    615   71.767364
## carol  346   76.666059
## dave   390   74.735465
## erin   556   50.538488

df.filter(like = "i")
## id weight height
## anna   168  102.91  188.67
## bob    615   71.76  155.90
## carol  346   76.66  171.38
## dave   390   74.73  173.15
## erin   556   50.53  183.08
```

```
df.filter(regex="ght$")
## weight  height
## anna  102.91  188.67
## bob  71.76  155.90
## carol  76.66  171.38
## dave  74.73  173.15
## erin  50.53  183.08

df.filter(like="o", axis=0)
##  id  weight  height  date
## bob  615  71.76  155.90
    2022-02-02
## carol  346  76.66  171.38
    2022-02-03
```

pandas - DataFrame: adding columns

Indexing with assignment allows for in-place modification of a DataFrame, while `assign()` creates a new object (but is chainable).

```
df['student'] = [True, True, True,
    False, None]
df['age'] = [19, 22, 25, None, None
    ]
df

## id weight height date student
    age
## anna   168  102.91  188.67
    2022-02-01    True  19.0
## dave   390   74.7  173.15
    2022-02-04   False   NaN
## erin   556   50.5  183.08
    2022-02-05    None   NaN
```

```
df.assign(student = lambda x:
     np.where(x.student, "yes
    ", "no"),
rand = np.random.rand(5)
)

## id weight    height   date
     student  age  and
## anna   168  102.91  188.67
     2022-02-01     yes  19.0
      0.60
## bob    615   71.76  155.90
     2022-02-02     yes  22.0
      0.54
```

pandas - DataFrame: removing columns (and rows)

Columns can be dropped via the `drop()` method,

```
df.drop(['student'])
## KeyError: "['student'] not found in axis."

df.drop(['student'], axis=1)
## id weight height  date  age
## anna   168  102.91  188.67  2022-02-01  19.0
## bob    615   71.76  155.90  2022-02-02  22.0
## carol  346   76.66  171.38  2022-02-03  25.0
## dave   390   74.73  173.15  2022-02-04   NaN
## erin   556   50.53  183.08  2022-02-05   NaN
```

```
df.drop(columns = df.columns == "age")
## KeyError: '[False, False, False, False, False, True] not found in axis'

df.drop(columns = df.columns[df.columns == "age"])
## id  weight height  date  student
## anna   168  102.9  188.67  2022-02-01    True
## bob    615   71.7  155.9  2022-02-02    True
## carol  346   76.6  171.3  2022-02-03    True
```

pandas - DataFrame: row binds

DataFrames can have their rows joined via the the `concat()` function (`append()` is also available but deprecated),

```
df1 = pd.DataFrame(np.arange(6).
    reshape(3,2), columns=list("
    xy"))
df1
##    x  y
## 0  0  1
## 1  2  3
## 2  4  5

pd.concat([df1,df2])
##     x   y
## 0   0   1
## 1   2   3
## 2   4   5
## 0  12  11
## 1  10   9
## 2   8   7
```

```
df2 = pd.DataFrame(np.arange
    (12,6,-1).reshape(3,2),
    columns=list("xy"))
df2
##     x   y
## 0  12  11
## 1  10   9
## 2   8   7

pd.concat([df1.loc[:,["y","x"]],
    df2])
##     y   x
## 0   1   0
## 1   3   2
## 2   5   4
## 0  11  12
## 1   9  10
```

Missing value problems and native Nas

pandas - Series: missing values

pandas encodes missing values using NaN (mostly),

```
s = pd.Series({"anna": "A+", "bob": "B-", "carol": "C", "dave": "D+"},
index = ["erin","dave","carol","bob","anna"])
s
## erin     NaN
## dave      D+
## carol      C
## bob       B-
## anna      A+
## dtype: object

pd.isna(s) = ?

```

```
s = pd.Series({"anna": 97, "bob": 82, "carol": 75, "dave": 68},
index = ["erin","dave","carol","bob","anna"],dtype='int64')
s
## erin      NaN
## dave     68.0
## carol    75.0
## bob      82.0
## anna     97.0
## dtype: float64

pd.isna(s) = ?

```

pandas - Series: aside - why np.isna()?

```
s = pd.Series([1,2,3,None])
s
## 0    1.0
## 1    2.0
## 2    3.0
## 3    NaN
## dtype: float64
```

```
pd.isna(s)
## 0    False
## 1    False
## 2    False
## 3     True
## dtype: bool

s == np.nan
## 0    False
## 1    False
## 2    False
## 3    False
## dtype: bool

```

```
np.nan == np.nan
## What about this?

np.nan != np.nan
## And this?

```

pandas - Series: native NAs

Recent versions of pandas have attempted to adopt a more native missing value, particularly for integer and boolean types,

```
pd.Series([1,2,3,None])
## 0    1.0
## 1    2.0
## 2    3.0
## 3    NaN
## dtype: float64

pd.Series([True,False,None])
## 0     True
## 1    False
## 2     None
## dtype: object

```

```
pd.isna( pd.Series([1,2,3,None]) )
## 0    False
## 1    False
## 2    False
## 3     True
## dtype: bool

pd.isna( pd.Series([True,False, None]) )
## 0    False
## 1    False
## 2     True
## dtype: bool

```