Basic Data Processing with Pandas

# The Series Data Structure

import pandas as pd

pd.Series?

Init signature: pd.Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)

Docstring:

One-dimensional ndarray with axis labels (including time series).

Labels need not be unique but must be any hashable type. The object

supports both integer- and label-based indexing and provides a host of

methods for performing operations involving the index. Statistical

methods from ndarray have been overridden to automatically exclude

missing data (currently represented as NaN)

Operations between Series (+, -, /, \*, \*\*) align values based on their associated index values-- they need not be the same length. The result

index will be the sorted union of the two indexes.

Parameters

----------

data : array-like, dict, or scalar value

Contains **data stored in Series**

index : array-like or Index (1d)

**Values must be unique and hashable, same length as data**. Index

object (or other iterable of same length as data) Will default to

RangeIndex(len(data)) if not provided. If both a dict and index

sequence are used, the index will override the keys found in the

dict.

dtype : numpy.dtype or None

If None, dtype will be inferred

copy : boolean, default False

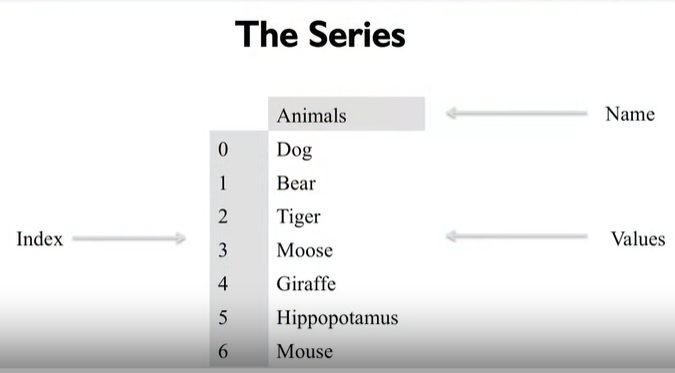
Copy input data

File: /opt/conda/lib/python3.6/site-packages/pandas/core/series.py

Type: type

(series: core data structures in pandas

Between a list and a dictionary)



animals = ['Tiger', 'Bear', 'Moose']

pd.**Series**(animals)

0 Tiger

1 Bear

2 Moose

dtype: object

numbers = [1, 2, 3] # list into Series

pd.Series(numbers)

0 1

1 2

2 3

dtype: int64

animals = ['Tiger', 'Bear', None] # contains missing data

pd.Series(animals)

0 Tiger

1 Bear

2 None

dtype: object

numbers = [1, 2, None] # NaN: not a number

pd.Series(numbers)

0 1.0

1 2.0

2 NaN

dtype: float64

import numpy as np

np.nan == None # NaN is not None

False

np.nan == np.nan # NaN cannot compare to itself

False

np.**isnan**(np.nan) # use the function of comparion to compare

True

***Method1:***

sports = {'Archery': 'Bhutan',

'Golf': 'Scotland',

'Sumo': 'Japan',

'Taekwondo': 'South Korea'} # set the name of index either

s = pd.Series(sports)

s

Archery Bhutan

Golf Scotland

Sumo Japan

Taekwondo South Korea

dtype: object

s.**index**

Index(['Archery', 'Golf', 'Sumo', 'Taekwondo'], dtype='object')

***Method2:***

s = pd.Series(['Tiger', 'Bear', 'Moose'], index=['India', 'America', 'Canada']) # must keep the same number

s

India Tiger

America Bear

Canada Moose

dtype: object

sports = {'Archery': 'Bhutan',

'Golf': 'Scotland',

'Sumo': 'Japan',

'Taekwondo': 'South Korea'}

s = pd.Series(sports, index=['Golf', 'Sumo', 'Hockey']) # NaN if no corresponding value

s

Golf Scotland

Sumo Japan

Hockey NaN

dtype: object

# Querying a Series

A panda.Series can be queried, either by the index position or the index label.

sports = {'Archery': 'Bhutan',

'Golf': 'Scotland',

'Sumo': 'Japan',

'Taekwondo': 'South Korea'}

s = pd.Series(sports)

s

Archery Bhutan

Golf Scotland

Sumo Japan

Taekwondo South Korea

dtype: object

# index operator

s.**iloc**[3] # locate with sequence

'South Korea'

s.**loc**['Golf'] # locate with index name

'Scotland'

(iloc and loc are not methods but attributes)

s[3]

'South Korea'

s['Golf']

'Scotland'

sports = {99: 'Bhutan',

100: 'Scotland',

101: 'Japan',

102: 'South Korea'}

s = pd.Series(sports)

s

99 Bhutan

100 Scotland

101 Japan

102 South Korea

dtype: object

s[0] #This won't call s.iloc[0] as one might expect, it generates an error instead

s[101]

'Japan'

s = pd.Series([100.00, 120.00, 101.00, 3.00])

s

0 100.0

1 120.0

2 101.0

3 3.0

dtype: float64

total = 0

for item in s: # iterating over

total+=item

print(total)

324.0

import numpy as np

total = np.sum(s)

print(total)

324.0

#this creates a big series of random numbers

s = pd.Series(np.random.randint(0,1000,10000))

s.head()

0 34

1 755

2 922

3 613

4 645

dtype: int64

**len**(s)

10000

%%**timeit** -n 100

# run 100 times and get how long it takes in average

summary = 0

for item in s:

summary+=item

1.83 ms ± 166 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

%%timeit -n 100

summary = np.sum(s) # vectorization

The slowest run took 6.08 times longer than the fastest. This could mean that an intermediate result is being cached.

181 µs ± 181 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

s+=2 #adds two to each item in s using broadcasting

s.head()

0 36

1 757

2 924

3 615

4 647

dtype: int64

for label, value in s.**iteritems()**:

s.set\_value(label, value+2) # same result, procedural way

s.head()

0 38

1 759

2 926

3 617

4 649

dtype: int64

%%timeit -n 10

s = pd.Series(np.random.randint(0,1000,10000))

s+=2

The slowest run took 24.70 times longer than the fastest. This could mean that an intermediate result is being cached.

1.14 ms ± 2.16 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

%%timeit -n 10

s = pd.Series(np.random.randint(0,1000,10000))

for label, value in s.iteritems():

s.set\_value(label, value+2)

50.1 ms ± 4.53 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

s = pd.Series([1, 2, 3])

s.loc['Animal'] = 'Bears' # loc not only search but also add new data

s

0 1

1 2

2 3

Animal Bears

dtype: object

***example of not unique indexes:***

original\_sports = pd.Series({'Archery': 'Bhutan',

'Golf': 'Scotland',

'Sumo': 'Japan',

'Taekwondo': 'South Korea'})

cricket\_loving\_countries = pd.Series(['Australia',

'Barbados',

'Pakistan',

'England'],

index=['Cricket',

'Cricket',

'Cricket',

'Cricket'])

all\_countries = original\_sports.**append**(cricket\_loving\_countries)

original\_sports

Archery Bhutan

Golf Scotland

Sumo Japan

Taekwondo South Korea

dtype: object

cricket\_loving\_countries

Cricket Australia

Cricket Barbados

Cricket Pakistan

Cricket England

dtype: object

all\_countries

Archery Bhutan

Golf Scotland

Sumo Japan

Taekwondo South Korea

Cricket Australia

Cricket Barbados

Cricket Pakistan

Cricket England

dtype: object

all\_countries.loc['Cricket']

Cricket Australia

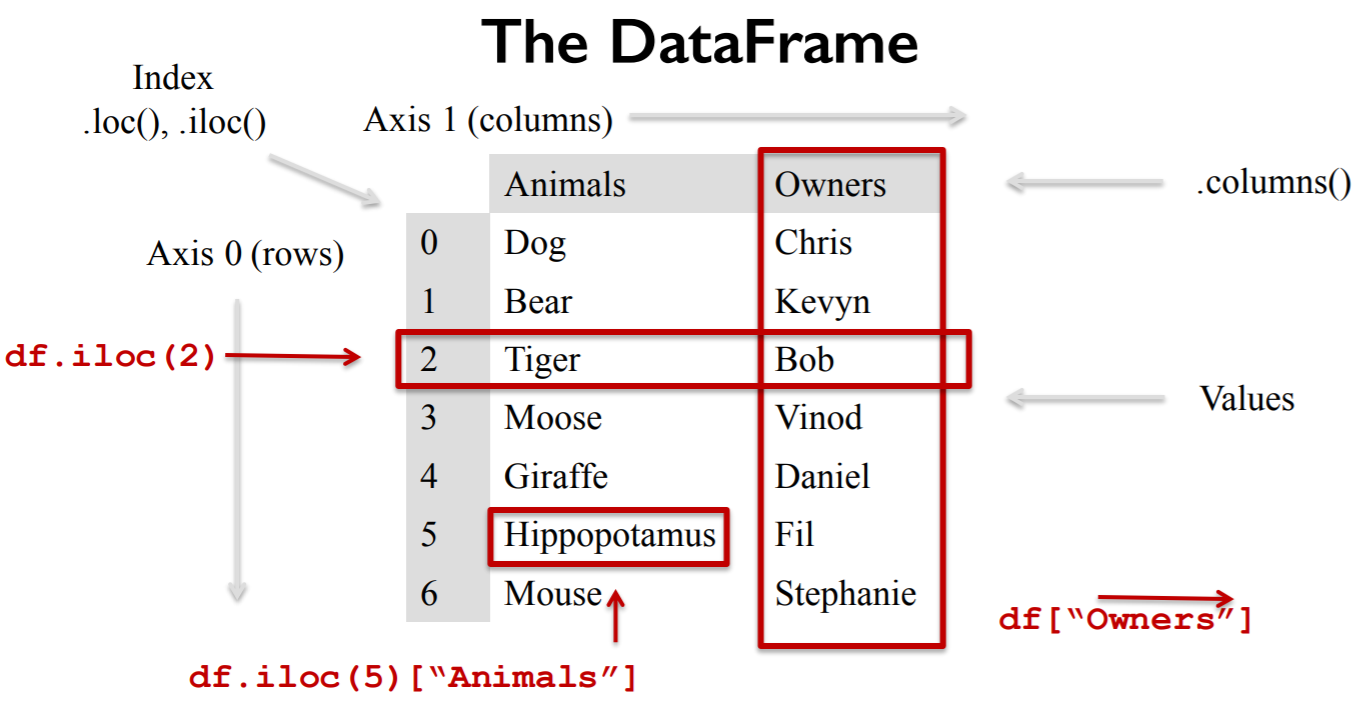
Cricket Barbados

Cricket Pakistan

Cricket England

dtype: object

# The DataFrame Data Structure



import pandas as pd

purchase\_1 = pd.Series({'Name': 'Chris',

'Item Purchased': 'Dog Food',

'Cost': 22.50})

purchase\_2 = pd.Series({'Name': 'Kevyn',

'Item Purchased': 'Kitty Litter',

'Cost': 2.50})

purchase\_3 = pd.Series({'Name': 'Vinod',

'Item Purchased': 'Bird Seed',

'Cost': 5.00})

df = pd.**DataFrame**([purchase\_1, purchase\_2, purchase\_3], index=['Store 1', 'Store 1', 'Store 2'])

df.head()



df.loc['Store 2'] # there is an index called 'Store 2'

Cost 5

Item Purchased Bird Seed

Name Vinod

Name: Store 2, dtype: object

**type**(df.loc['Store 2']) # returns ‘Series’

pandas.core.series.Series

df.loc['Store 1'] # there is an index called 'Store 1'



df.loc['Store 1', 'Cost'] # there is an index called 'Store 1', then get specific to a column

Store 1 22.5

Store 1 2.5

Name: Cost, dtype: float64

df.**T #** transposed



df.T.loc['Cost'] # after transpose, there is an index called 'Cost'

Store 1 22.5

Store 1 2.5

Store 2 5

Name: Cost, dtype: object

df['Cost'] # no index with the name of 'Cost', choose a column directly

Store 1 22.5

Store 1 2.5

Store 2 5.0

Name: Cost, dtype: float64

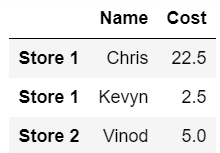
df.loc['Store 1']['Cost'] # cost of store 1

Store 1 22.5

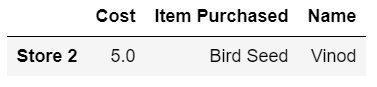
Store 1 2.5

Name: Cost, dtype: float64

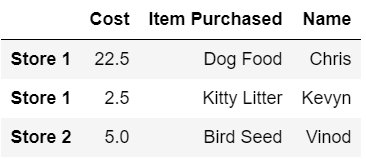
df.loc[ : , ['Name', 'Cost']] # name and cost of all indexes



df.drop('Store 1') # keep 'Store 1' away (only from this copy)



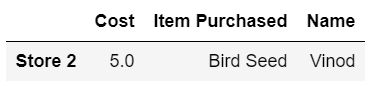
df # origin still unchanged



copy\_df = df.copy()

copy\_df = copy\_df.drop('Store 1') # assign to

copy\_df



copy\_df.drop?

Signature: copy\_df.drop(labels, axis=0, level=None, inplace=False, errors='raise')

Docstring:

Return new object with labels in requested axis removed.

Parameters

----------

labels : single label or list-like

axis : int or axis name

level : int or level name, default None

For MultiIndex

inplace : bool, default False

If True, do operation inplace and return None.

errors : {'ignore', 'raise'}, default 'raise'

If 'ignore', suppress error and existing labels are dropped.

.. versionadded:: 0.16.1

Returns

-------

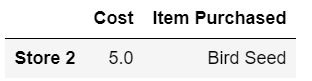
dropped : type of caller

File: /opt/conda/lib/python3.6/site-packages/pandas/core/generic.py

Type: method

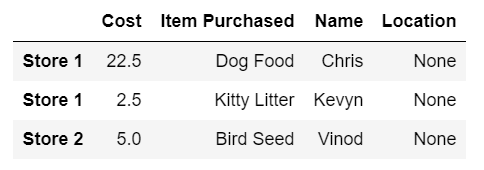
**del** copy\_df['Name'] # delete

copy\_df



df['Location'] = None # set to None but keep values

df



# Dataframe Indexing and Loading

costs = df['Cost']

costs

Store 1 22.5

Store 1 2.5

Store 2 5.0

Name: Cost, dtype: float64

costs+=2

costs

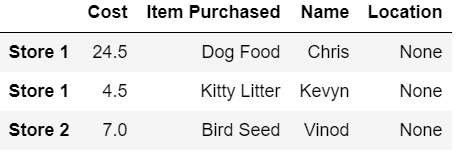
Store 1 24.5

Store 1 4.5

Store 2 7.0

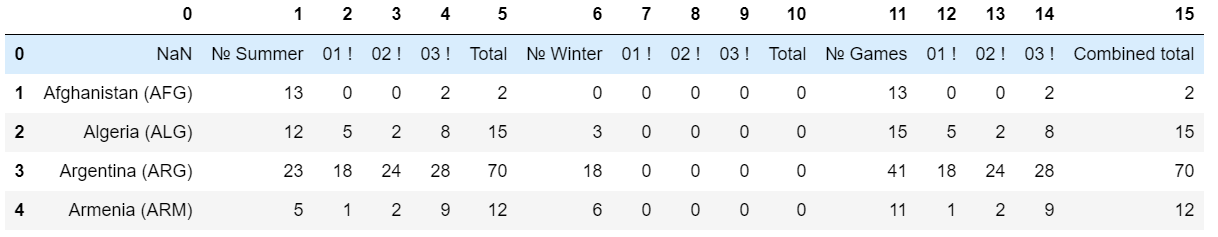
Name: Cost, dtype: float64

df



df = pd.**read\_csv**('olympics.csv')

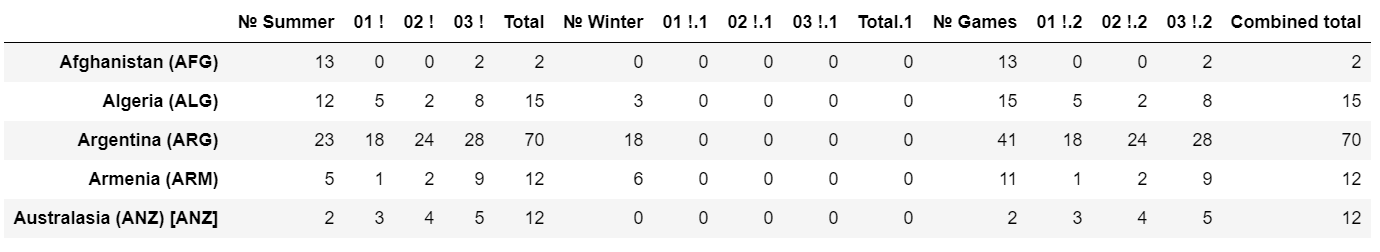
df.head()



df = pd.read\_csv('olympics.csv', index\_col = 0, skiprows=1)

# make the first column indexes, skip one row set the correct column names

df.head()



df.columns

Index(['№ Summer', '01 !', '02 !', '03 !', 'Total', '№ Winter', '01 !.1',

'02 !.1', '03 !.1', 'Total.1', '№ Games', '01 !.2', '02 !.2', '03 !.2',

'Combined total'],

dtype='object')

for col in df.columns: # rename(replace) columns

if col[:2]=='01':

df.rename(columns={col:'Gold' + col[4:]}, inplace=True) # inplace=True modify self

if col[:2]=='02':

if col[:2]=='02':

df.rename(columns={col:'Silver' + col[4:]}, inplace=True)

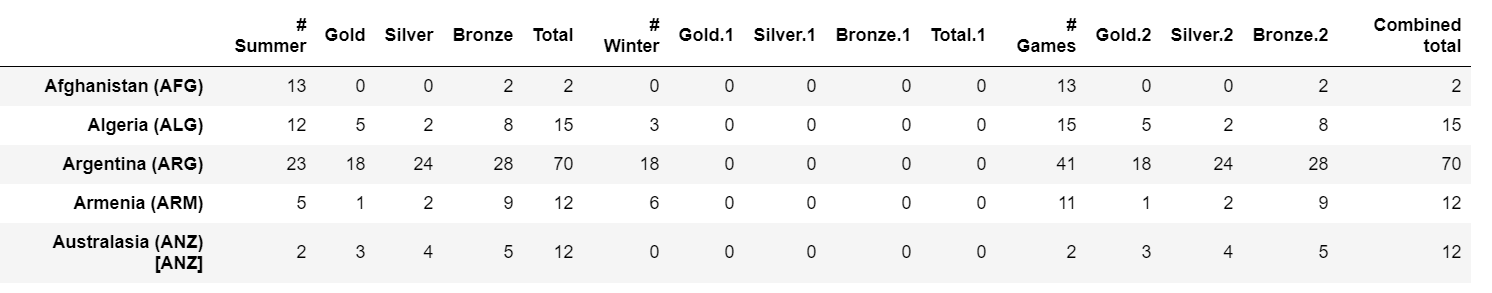
if col[:2]=='03':

df.rename(columns={col:'Bronze' + col[4:]}, inplace=True)

if col[:1]=='№':

df.rename(columns={col:'#' + col[1:]}, inplace=True)

df.head()



# Querying a DataFrame

**(Boolean masking: use conditions to filter data and only keep True values)**

df['Gold'] > 0

Afghanistan (AFG) False

Algeria (ALG) True

Argentina (ARG) True

Armenia (ARM) True

Australasia (ANZ) [ANZ] True

Australia (AUS) [AUS] [Z] True

Austria (AUT) True

Azerbaijan (AZE) True

Bahamas (BAH) True

Bahrain (BRN) False

Barbados (BAR) [BAR] False

Belarus (BLR) True

Belgium (BEL) True

Bermuda (BER) False

Bohemia (BOH) [BOH] [Z] False

Botswana (BOT) False

Brazil (BRA) True

British West Indies (BWI) [BWI] False

Bulgaria (BUL) [H] True

Burundi (BDI) True

Cameroon (CMR) True

Canada (CAN) True

Chile (CHI) [I] True

China (CHN) [CHN] True

Colombia (COL) True

Costa Rica (CRC) True

Ivory Coast (CIV) [CIV] False

Croatia (CRO) True

Cuba (CUB) [Z] True

Cyprus (CYP) False

...

Sri Lanka (SRI) [SRI] False

Sudan (SUD) False

Suriname (SUR) [E] True

Sweden (SWE) [Z] True

Switzerland (SUI) True

Syria (SYR) True

Chinese Taipei (TPE) [TPE] [TPE2] True

Tajikistan (TJK) False

Tanzania (TAN) [TAN] False

Thailand (THA) True

Togo (TOG) False

Tonga (TGA) False

Trinidad and Tobago (TRI) [TRI] True

Tunisia (TUN) True

Turkey (TUR) True

Uganda (UGA) True

Ukraine (UKR) True

United Arab Emirates (UAE) True

United States (USA) [P] [Q] [R] [Z] True

Uruguay (URU) True

Uzbekistan (UZB) True

Venezuela (VEN) True

Vietnam (VIE) False

Virgin Islands (ISV) False

Yugoslavia (YUG) [YUG] True

Independent Olympic Participants (IOP) [IOP] False

Zambia (ZAM) [ZAM] False

Zimbabwe (ZIM) [ZIM] True

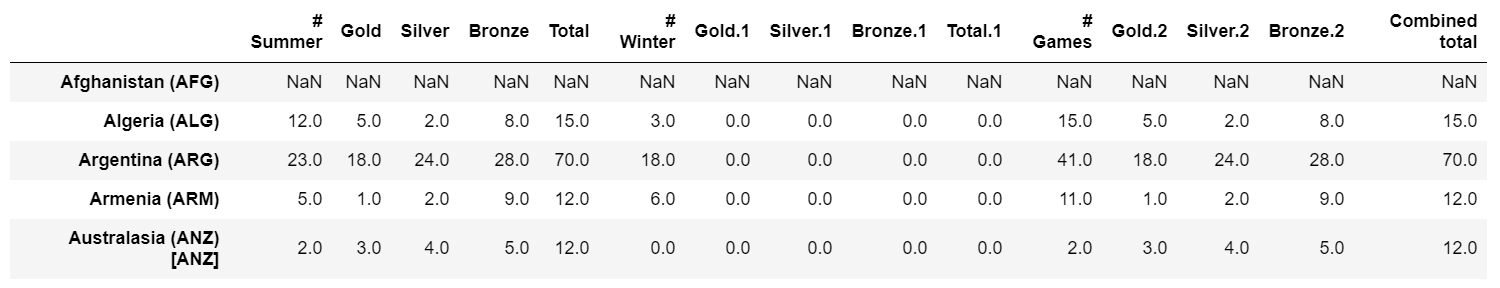
Mixed team (ZZX) [ZZX] True

Totals True

Name: Gold, dtype: bool

only\_gold = df.where(df['Gold'] > 0)

only\_gold.head()



only\_gold['Gold'].count()

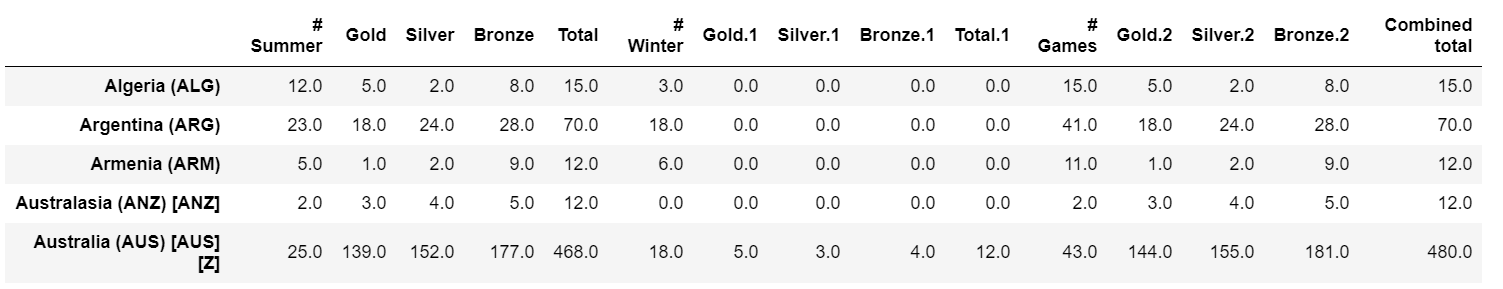
100

df['Gold'].count()

147

only\_gold = only\_gold.**dropna**() # drop all NaN

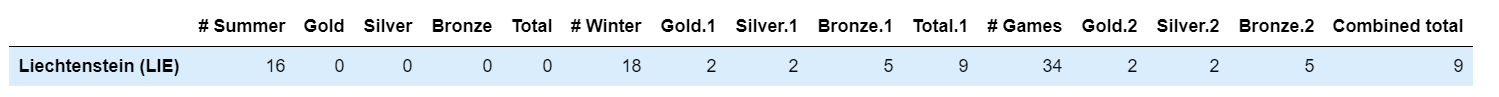
only\_gold.head()



len(df[(df['Gold'] > 0) | (df['Gold.1'] > 0)])

101

df[(df['Gold.1'] > 0) & (df['Gold'] == 0)]



***Example:***

purchase\_1 = pd.Series({'Name': 'Chris',

'Item Purchased': 'Dog Food',

'Cost': 22.50})

purchase\_2 = pd.Series({'Name': 'Kevyn',

'Item Purchased': 'Kitty Litter',

'Cost': 2.50})

purchase\_3 = pd.Series({'Name': 'Vinod',

'Item Purchased': 'Bird Seed',

'Cost': 5.00})

df = pd.DataFrame([purchase\_1, purchase\_2, purchase\_3], index=['Store 1', 'Store 1', 'Store 2'])

df['Name'][df['Cost']>3]

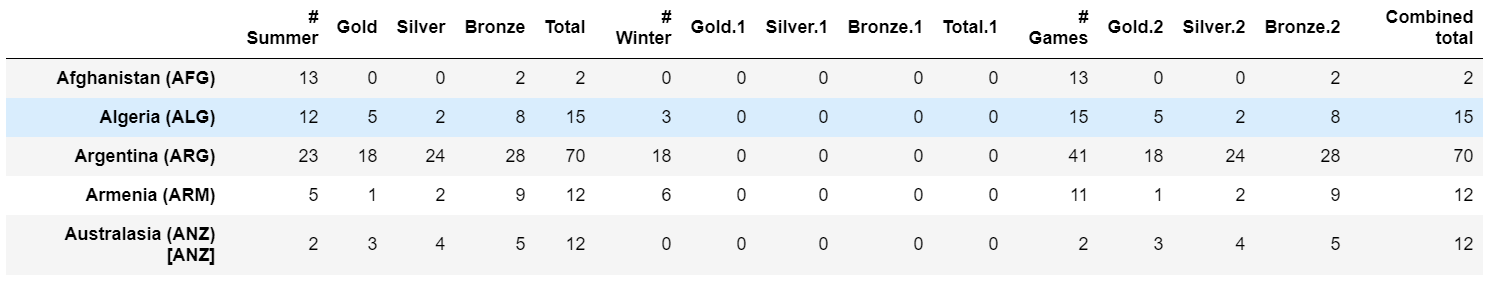
Store 1 Chris

Store 2 Vinod

Name: Name, dtype: object

# Indexing Dataframes

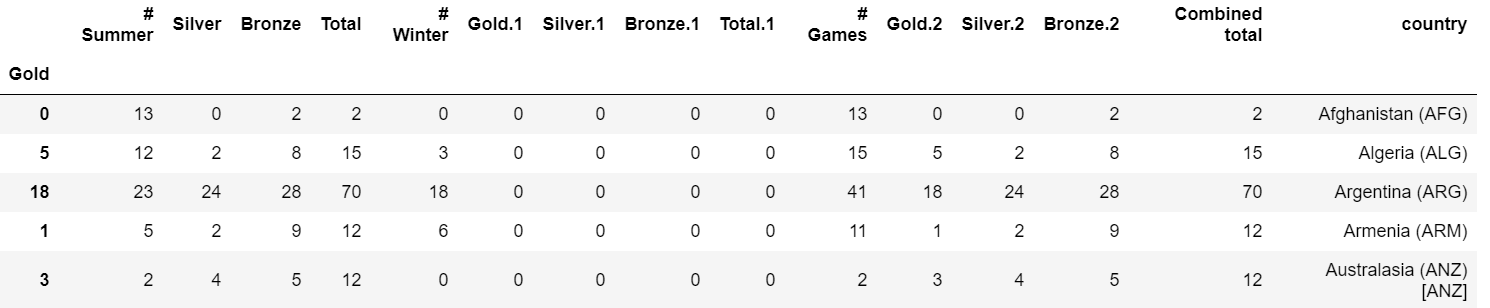
df.head()



df['country'] = df.index # preserve country into a new column

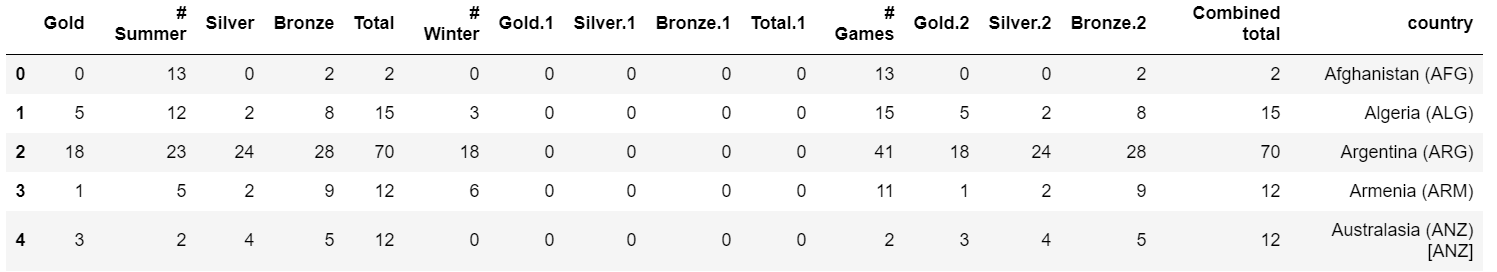
df = df.**set\_index**('Gold')

df.head()



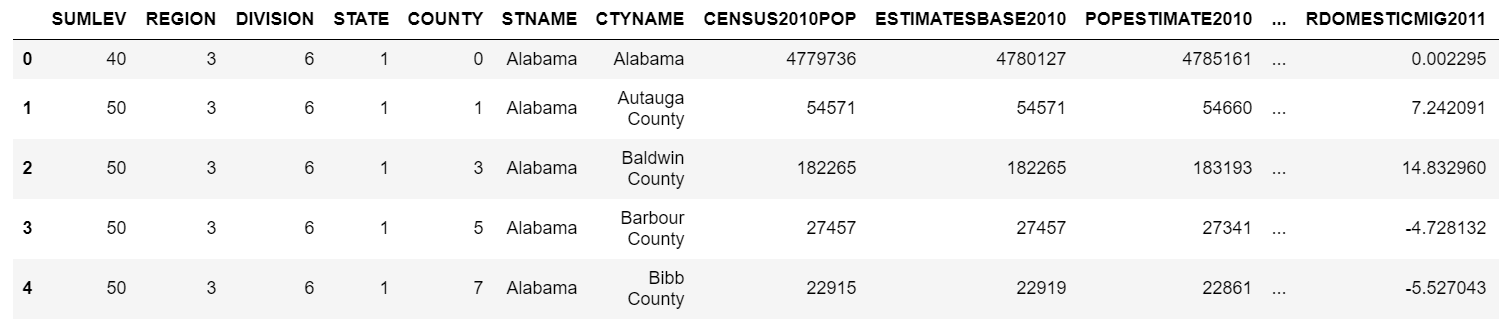
df = df**.reset\_index**()

df.head()



df = pd.read\_csv('census.csv')

df.head()

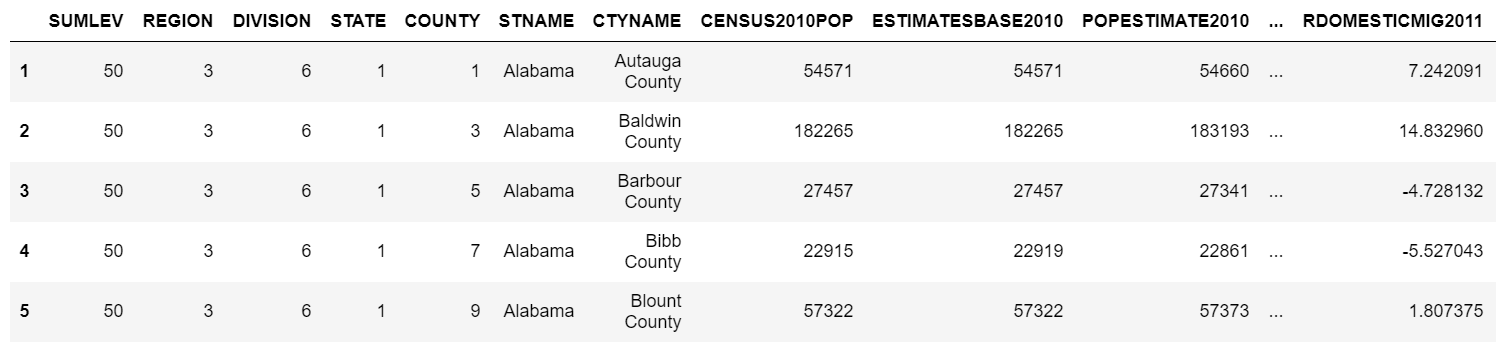


df['SUMLEV'].unique() # show different values

array([40, 50])

df=df[df['SUMLEV'] == 50]

df.head()



columns\_to\_keep = ['STNAME',

'CTYNAME',

'BIRTHS2010',

'BIRTHS2011',

'BIRTHS2012',

'BIRTHS2013',

'BIRTHS2014',

'BIRTHS2015',

'POPESTIMATE2010',

'POPESTIMATE2011',

'POPESTIMATE2012',

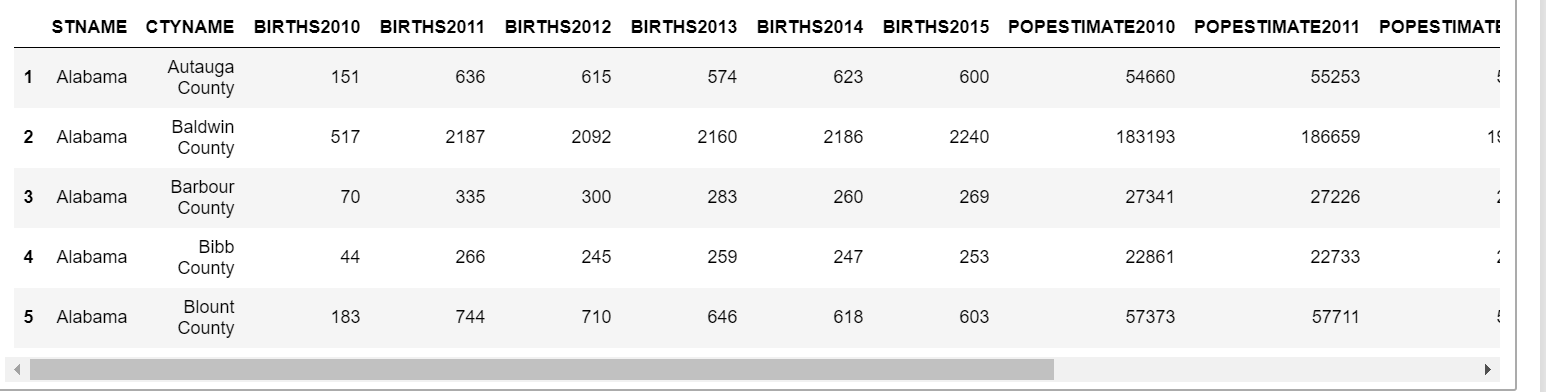
'POPESTIMATE2013',

'POPESTIMATE2014',

'POPESTIMATE2015']

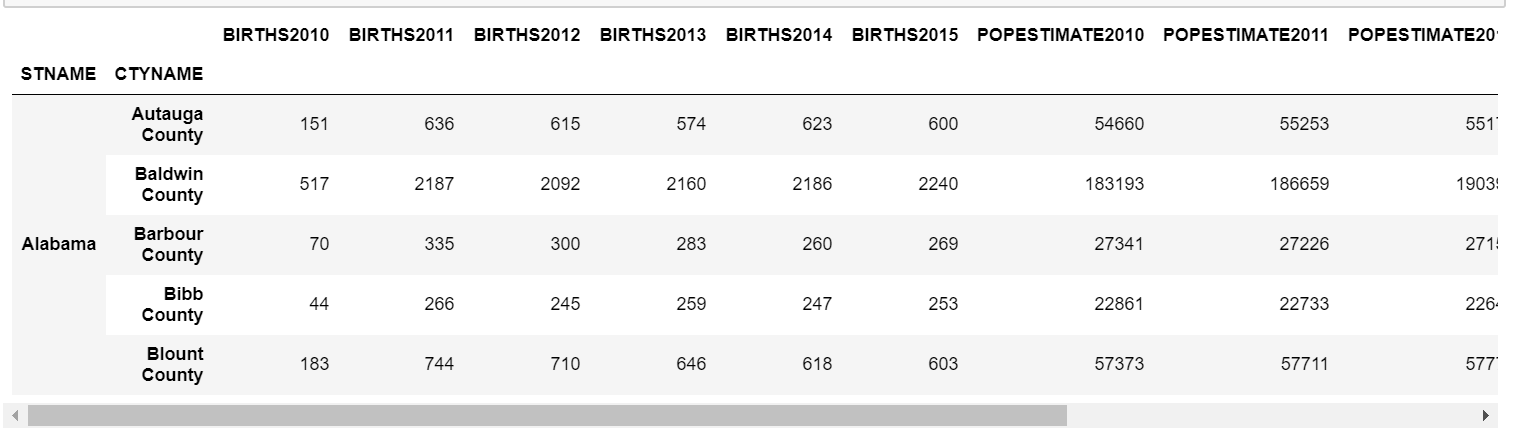
df = df[columns\_to\_keep] # keep only parts of columns, assign

df.head()



df = df.set\_index(['STNAME', 'CTYNAME']) # multiindex

df.head()



df.loc['Michigan', 'Washtenaw County']

BIRTHS2010 977

BIRTHS2011 3826

BIRTHS2012 3780

BIRTHS2013 3662

BIRTHS2014 3683

BIRTHS2015 3709

POPESTIMATE2010 345563

POPESTIMATE2011 349048

POPESTIMATE2012 351213

POPESTIMATE2013 354289

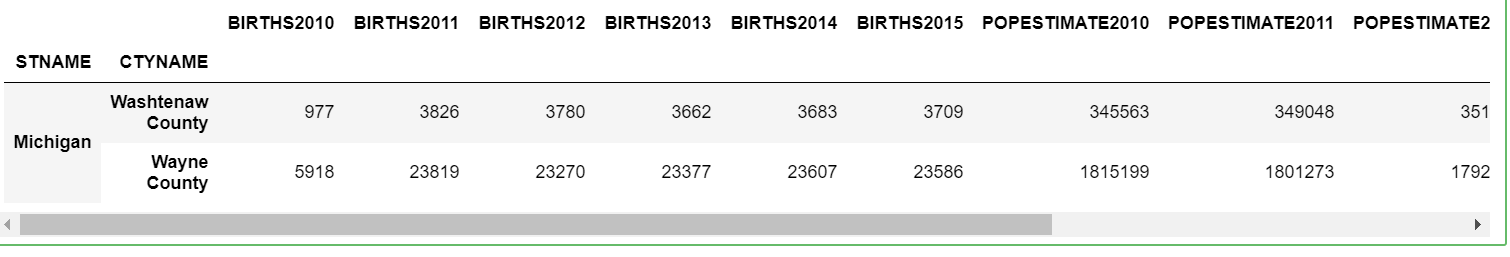
POPESTIMATE2014 357029

POPESTIMATE2015 358880

Name: (Michigan, Washtenaw County), dtype: int64

df.loc[ [('Michigan', 'Washtenaw County'),

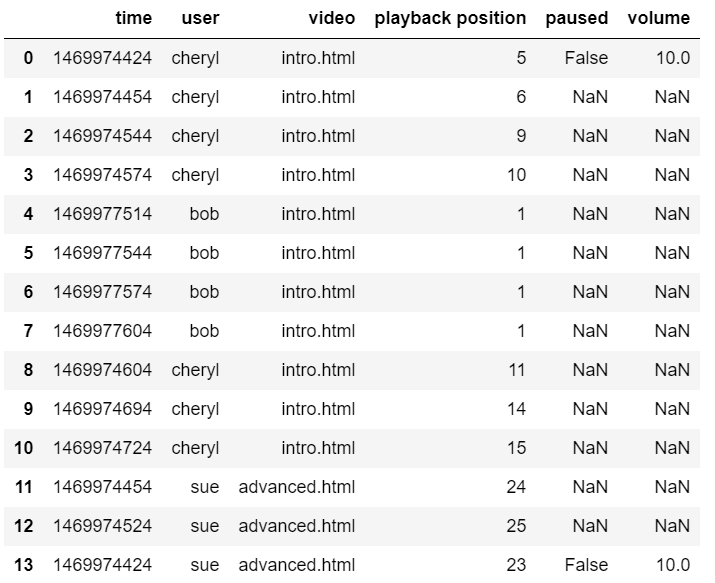
('Michigan', 'Wayne County')] ]

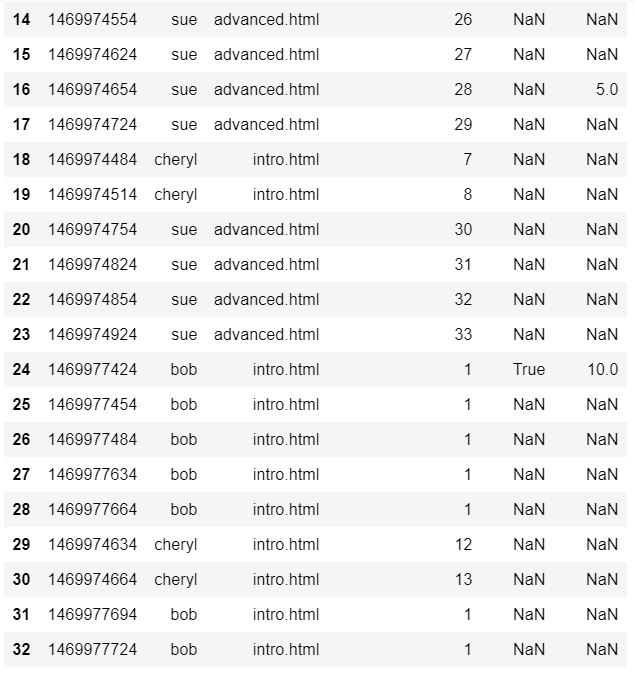


# Missing values

df = pd.read\_csv('log.csv')

df





df.fillna?

Signature: df.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None, \*\*kwargs)

Docstring:

Fill NA/NaN values using the specified method

Parameters

----------

value : scalar, dict, Series, or DataFrame

Value to use to fill holes (e.g. 0), alternately a

dict/Series/DataFrame of values specifying which value to use for

each index (for a Series) or column (for a DataFrame). (values not

in the dict/Series/DataFrame will not be filled). This value cannot

be a list.

method : {'backfill', 'bfill', 'pad', 'ffill', None}, default None

Method to use for filling holes in reindexed Series

pad / ffill: propagate last valid observation forward to next valid

backfill / bfill: use NEXT valid observation to fill gap

axis : {0 or 'index', 1 or 'columns'}

inplace : boolean, default False

If True, fill in place. Note: this will modify any

other views on this object, (e.g. a no-copy slice for a column in a

DataFrame).

limit : int, default None

If method is specified, this is the maximum number of consecutive

NaN values to forward/backward fill. In other words, if there is

a gap with more than this number of consecutive NaNs, it will only

be partially filled. If method is not specified, this is the

maximum number of entries along the entire axis where NaNs will be

filled.

downcast : dict, default is None

a dict of item->dtype of what to downcast if possible,

or the string 'infer' which will try to downcast to an appropriate

equal type (e.g. float64 to int64 if possible)

See Also

--------

reindex, asfreq

Returns

-------

filled : DataFrame

File: /opt/conda/lib/python3.6/site-packages/pandas/core/frame.py

Type: method

df = df.set\_index('time')

df = df.sort\_index()

df

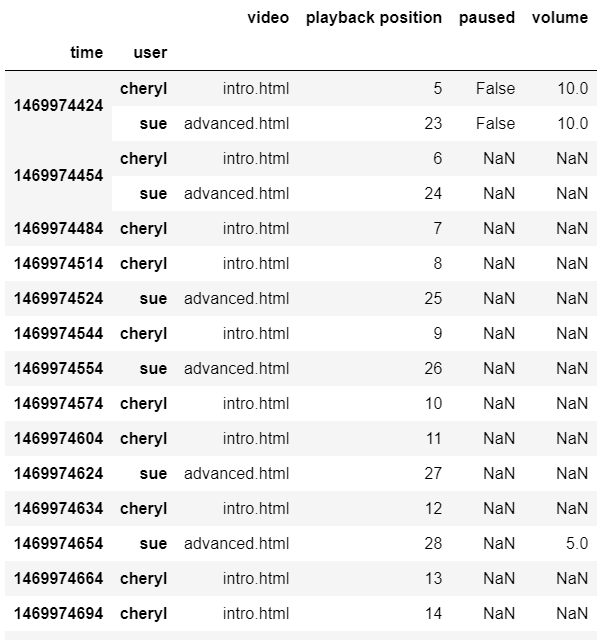


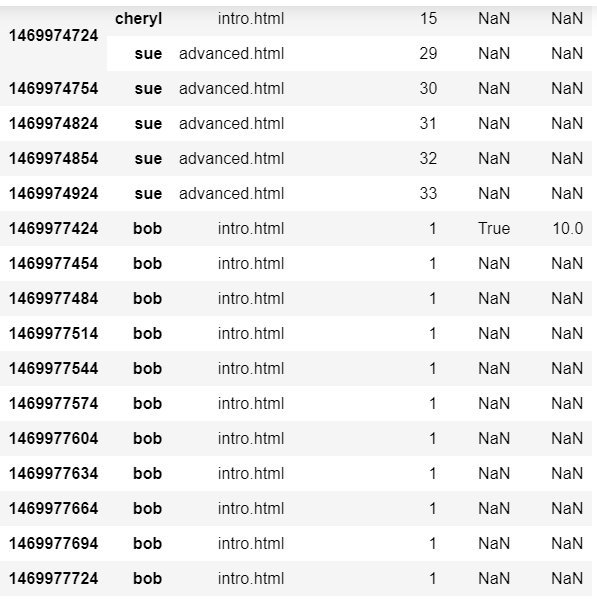


df = df.reset\_index()

df = df.set\_index(['time', 'user'])

df





df = df.fillna(method='ffill') # use the previous observation

df

