Lecture 5-1

Pandas: Indexing, Arithmetic, Missing Values

Week 5 Monday

Miles Chen, PhD

Based on Wes McKinney's Python for Data Analysis and the Pandas Documentation

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

Series that we will use as examples

In [2]:
```
# note that the value after the decimal place corresponds to the letter position.
# i.e. 1.4 corresponds to d, the fourth letter.
original1 = pd.Series([1.4, 2.3, 3.1, 4.2], index = ['d','c','a','b'])
original2 = pd.Series([2.2, 3.1, 1.3, 4.4], index = ['b','a','c','d'])
```

In [3]:
```
original1 # when you create a series, the original order of the index is preserved
```

Out[3]:
```
d    1.4
c    2.3
a    3.1
b    4.2
dtype: float64
```

In [4]:
```
# making a DataFrame with multiple series with the same index preserves the index order
pd.DataFrame({"x":original1, "x2": original1 * 2})
```

Out[4]:

	x	x2
d	1.4	2.8
c	2.3	4.6
a	3.1	6.2
b	4.2	8.4

In [5]:
```
original2 # note that original1 and original2 have different index orders
```

Out[5]:
```
b    2.2
a    3.1
c    1.3
d    4.4
dtype: float64
```

In [6]:
```
# because original1 and original2 have index in different order, Pandas will sort the index before putting them toget
df = pd.DataFrame({"x":original1, "y": original2})
df
```

Out[6]:

	x	y
a	3.1	3.1
b	4.2	2.2
c	2.3	1.3
d	1.4	4.4

In [7]:
```
original1.index # the index of original1 is the letters d, c, a, b in a tuple-like object
```

Out[7]:
```
Index(['d', 'c', 'a', 'b'], dtype='object')
```

In [8]:
```
original1['d':'a'] # when slicing pandas uses the index order or original1
```

Out[8]:
```
d    1.4
c    2.3
a    3.1
dtype: float64
```

In [9]:
```
df.index  # the index of df are the letters abcd in order
```

Out[9]:
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

In [10]:
```
df['a':'c'] # when slicing Pandas uses the index order of the DataFrame, which has been sorted
```

Out[10]:

	x	y
a	3.1	3.1
b	4.2	2.2
c	2.3	1.3

Rearranging value

Both Series and DataFrames have the `.sort_index()` and `.sort_values()` methods which can be used to rearrange the value.

In [11]:
```
original2
```

Out[11]:
```
b    2.2
a    3.1
c    1.3
d    4.4
dtype: float64
```

In [12]:
```
original2.sort_index()
```

Out[12]:
```
a    3.1
b    2.2
c    1.3
d    4.4
dtype: float64
```

In [13]:
```
original2.sort_values()
```

Out[13]:
```
c    1.3
b    2.2
a    3.1
d    4.4
dtype: float64
```

In [14]:
```
df
```

Out[14]:

	x	y
a	3.1	3.1
b	4.2	2.2
c	2.3	1.3
d	1.4	4.4

In [15]:
```
df.sort_values(by = "x", ascending = False)
```

Out[15]:

	x	y
b	4.2	2.2
a	3.1	3.1
c	2.3	1.3
d	1.4	4.4

Changing the Index

The index of a Pandas Series or Pandas DataFrame is immutable and cannot be modified.

However, if you want to change the index of a series or dataframe, you can define a new index and replace the existing index of the series/DataFrame.

In [16]:
```
original1.index = range(4) # I replace the index of the series with this range object.
```

In [17]:
```
original1
```

Out[17]:
```
0    1.4
1    2.3
2    3.1
3    4.2
dtype: float64
```

In [18]:
```
original1.index # We can see this has automatically become a RangeIndex object
```

Out[18]:
```
RangeIndex(start=0, stop=4, step=1)
```

In [19]:
```
original1[1]
```

Out[19]:
```
np.float64(2.3)
```

In [20]:
```
original1.loc[1] # behaves the same as above
```

Out[20]:
```
np.float64(2.3)
```

In [21]:
```
original1.iloc[1] # behaves the same as above because the range index starts at 0
```

Out[21]:
```
np.float64(2.3)
```

In [22]:
```
original1.index = range(1,5)
```

In [23]:
```
original1
```

Out[23]:
```
1    1.4
2    2.3
3    3.1
4    4.2
dtype: float64
```

In [24]:
```
original1[1]
```

Out[24]:
```
np.float64(1.4)
```

In [25]:
```
original1.loc[1]
```

Out[25]:
```
np.float64(1.4)
```

In [26]:
```
original1.iloc[1] # behavior is different because range index starts at 1
```

Out[26]:
```
np.float64(2.3)
```

In [27]:
```
original1['a'] # throws an error because 'a' is no longer part of the index and cannot be used to select values
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[27], line 1
----> 1 original1['a'] # throws an error because 'a' is no longer part of the index and cannot be used to select valu
es

File c:\Users\miles\.pyenv\pyenv-win\versions\3.12.5\Lib\site-packages\pandas\core\series.py:1121, in Series.__getite
m__(self, key)
   1118     return self._values[key]
   1120 elif key_is_scalar:
-> 1121     return self._get_value(key)
   1123 # Convert generator to list before going through hashable part
   1124 # (We will iterate through the generator there to check for slices)
   1125 if is_iterator(key):

File c:\Users\miles\.pyenv\pyenv-win\versions\3.12.5\Lib\site-packages\pandas\core\series.py:1237, in Series._get_val
ue(self, label, takeable)
   1234     return self._values[label]
   1236 # Similar to Index.get_value, but we do not fall back to positional
-> 1237 loc = self.index.get_loc(label)
   1239 if is_integer(loc):
   1240     return self._values[loc]

File c:\Users\miles\.pyenv\pyenv-win\versions\3.12.5\Lib\site-packages\pandas\core\indexes\range.py:417, in RangeInde
x.get_loc(self, key)
    415         raise KeyError(key) from err
    416 if isinstance(key, Hashable):
--> 417     raise KeyError(key)
    418 self._check_indexing_error(key)
    419 raise KeyError(key)

KeyError: 'a'
```

In [28]:
```
original1.index = ['a','b','c','d'] # be careful as no restrictions regarding the meaning of the index is applied.
# in the original 'a' was associated with 3.1. This index will associate it with 1.4
```

In [29]:
```
original1
```

Out[29]:
```
a    1.4
b    2.3
c    3.1
d    4.2
dtype: float64
```

In [30]:
```
original1['a']
```

Out[30]:
```
np.float64(1.4)
```

In [31]:
```
original1[0] # now that the index uses strings, you can index by position
```

```
C:\Users\miles\AppData\Local\Temp\ipykernel_2740\1303254227.py:1: FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a future version, integer keys will always be treated as labels (consistent with DataFrame behavior). To access a value by position, use `ser.iloc[pos]`
  original1[0] # now that the index uses strings, you can index by position
```

Out[31]:
```
np.float64(1.4)
```

In [32]:
```
original1.index = [1, 2, 3, 4, 5] # if the object you provide is of a different length, you get a value error
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[32], line 1
----> 1 original1.index = [1, 2, 3, 4, 5] # if the object you provide is of a different length, you get a value error

File c:\Users\miles\.pyenv\pyenv-win\versions\3.12.5\Lib\site-packages\pandas\core\generic.py:6313, in NDFrame.__setattr__(self, name, value)
   6311 try:
   6312     object.__getattribute__(self, name)
-> 6313     return object.__setattr__(self, name, value)
   6314 except AttributeError:
   6315     pass

File properties.pyx:69, in pandas._libs.properties.AxisProperty.__set__()

File c:\Users\miles\.pyenv\pyenv-win\versions\3.12.5\Lib\site-packages\pandas\core\generic.py:814, in NDFrame._set_axis(self, axis, labels)
    809 """
    810 This is called from the cython code when we set the `index` attribute
    811 directly, e.g. `series.index = [1, 2, 3]`.
    812 """
    813 labels = ensure_index(labels)
--> 814 self._mgr.set_axis(axis, labels)
    815 self._clear_item_cache()

File c:\Users\miles\.pyenv\pyenv-win\versions\3.12.5\Lib\site-packages\pandas\core\internals\managers.py:238, in BaseBlockManager.set_axis(self, axis, new_labels)
    236 def set_axis(self, axis: AxisInt, new_labels: Index) -> None:
    237     # Caller is responsible for ensuring we have an Index object.
--> 238     self._validate_set_axis(axis, new_labels)
    239     self.axes[axis] = new_labels

File c:\Users\miles\.pyenv\pyenv-win\versions\3.12.5\Lib\site-packages\pandas\core\internals\base.py:98, in DataManager._validate_set_axis(self, axis, new_labels)
     95     pass
     97 elif new_len != old_len:
---> 98     raise ValueError(
     99         f"Length mismatch: Expected axis has {old_len} elements, new "
    100         f"values have {new_len} elements"
    101     )

ValueError: Length mismatch: Expected axis has 4 elements, new values have 5 elements
```

In [33]:
```
# similarly you can change the index of a DataFrame by defining a new object and assigning it to the index.
df.index = ['j','k','l','m']
df
```

Out[33]:

	x	y
j	3.1	3.1
k	4.2	2.2
l	2.3	1.3
m	1.4	4.4

Reindexing

Reindexing is different from just defining a new index.

Reindexing takes a current Pandas object and creates a *new* Pandas object that *conforms* to the specified index.

Do not confuse reindexing with creating a new index for a dataframe object.

In [34]:
```
original = pd.Series([1.4, 2.3, 3.1, 4.2], index = ['d','c','a','b'])
```

In [35]:
```
original
```

Out[35]:
```
d    1.4
c    2.3
a    3.1
b    4.2
dtype: float64
```

In [36]:
```
newobj = original.reindex(['a','b','c','d','e']) # note this has an index value that doesn't exist in the original se
```

In [37]:
```
newobj  # takes the data in orignal and moves it so it conforms to the specified index
# values that do not exist for the new index get NaN
```

Out[37]:
```
a    3.1
b    4.2
c    2.3
d    1.4
e    NaN
dtype: float64
```

In [38]:
```
# if you don't want NaN, you can specify a fill_value
newobj2 = original.reindex(['a','b','c','d','e'], fill_value = 0)
newobj2
```

Out[38]:
```
a    3.1
b    4.2
c    2.3
d    1.4
e    0.0
dtype: float64
```

For ordered data like a time series, it might be desirable to fill values when reindexing

In [39]:
```
obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 3, 6])
obj3
```

Out[39]:
```
0      blue
3    purple
6    yellow
dtype: object
```

In [40]:
```
obj3.reindex(range(9))  # without any optional arguments, lots of missing values
```

Out[40]:
```
0      blue
1       NaN
2       NaN
3    purple
4       NaN
5       NaN
6    yellow
7       NaN
8       NaN
dtype: object
```

In [41]:
```
obj3.reindex(range(9), method='ffill')
# forward-fill pushes values 'forward' until a new value is encountered
```

Out[41]:
```
0      blue
1      blue
2      blue
3    purple
4    purple
5    purple
6    yellow
7    yellow
8    yellow
dtype: object
```

In [42]:
```
obj3.reindex(range(9), method='bfill')
# back-fill works in the opposite direction
# there was no value at index 8 so, NaNs get filled in
```

Out[42]:
```
0      blue
1    purple
2    purple
3    purple
4    yellow
5    yellow
6    yellow
7       NaN
8       NaN
dtype: object
```

Date Ranges as Index

In [43]:
```
# we specify the creation of a date_index using the date_range function
# freq = 'D' creates Daily values
date_index = pd.date_range('1/1/2010', periods=6, freq='D')
date_index
```

Out[43]:
```
DatetimeIndex(['2010-01-01', '2010-01-02', '2010-01-03', '2010-01-04',
               '2010-01-05', '2010-01-06'],
              dtype='datetime64[ns]', freq='D')
```

In [44]:
```
# we create a DataFrame with the date index
df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]}, index=date_index)
df2
```

Out[44]:

	prices
2010-01-01	100.0
2010-01-02	101.0
2010-01-03	NaN
2010-01-04	100.0
2010-01-05	89.0
2010-01-06	88.0

In [45]:
```
# we create a DataFrame with the date index
df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]}, index=date_index)
df2
```

Out[45]:

	prices
2010-01-01	100.0
2010-01-02	101.0
2010-01-03	NaN
2010-01-04	100.0
2010-01-05	89.0
2010-01-06	88.0

In [46]:
```
date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')  # a new date index
df2.reindex(date_index2)
```

Out[46]:

	prices
2009-12-29	NaN
2009-12-30	NaN
2009-12-31	NaN
2010-01-01	100.0
2010-01-02	101.0
2010-01-03	NaN
2010-01-04	100.0
2010-01-05	89.0
2010-01-06	88.0
2010-01-07	NaN

In [47]:

```
df2.reindex(date_index2, method = 'bfill')
# The value for Jan 3 isn't filled in because that NaN was not created by the reindexing process
# The NaN already existed in the data.
```

Out[47]:

	prices
2009-12-29	100.0
2009-12-30	100.0
2009-12-31	100.0
2010-01-01	100.0
2010-01-02	101.0
2010-01-03	NaN
2010-01-04	100.0
2010-01-05	89.0
2010-01-06	88.0
2010-01-07	NaN

`.reindex()` vs `.loc()`

If you don't need to fill in any missing info, then `.reindex()` and `.loc()` work very similarly. If the new index will have values that don't exist in the current index, you need to use `reindex`.

In [48]:
```
obj5 = pd.DataFrame({'val':[1.4, 2.3, 3.1, 4.2]}, index = ['d','c','a','b'])
obj5
```

Out[48]:

	val
d	1.4
c	2.3
a	3.1
b	4.2

In [49]:
```
obj5.reindex(['a','b','c','d'])
```

Out[49]:

	val
a	3.1
b	4.2
c	2.3
d	1.4

In [50]:
```
obj5.loc[['a','b','c','d']] # works the same as reindex
```

Out[50]:

	val
a	3.1
b	4.2
c	2.3
d	1.4

In [51]:
```
obj5.reindex(['a','b','c','d','e'])
```

Out[51]:

	val
a	3.1
b	4.2
c	2.3
d	1.4
e	NaN

In [52]:
```
obj5.loc[['a','b','c','d','e']]  # .loc() returns an error if you give an entry in the index that doesn't exist
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[52], line 1
----> 1 obj5.loc[['a','b','c','d','e']]  # .loc() returns an error if you give an entry in the index that doesn't exi
st

File c:\Users\miles\.pyenv\pyenv-win\versions\3.12.5\Lib\site-packages\pandas\core\indexing.py:1191, in _LocationInde
xer.__getitem__(self, key)
   1189 maybe_callable = com.apply_if_callable(key, self.obj)
   1190 maybe_callable = self._check_deprecated_callable_usage(key, maybe_callable)
-> 1191 return self._getitem_axis(maybe_callable, axis=axis)

File c:\Users\miles\.pyenv\pyenv-win\versions\3.12.5\Lib\site-packages\pandas\core\indexing.py:1420, in _LocIndexer._
getitem_axis(self, key, axis)
   1417     if hasattr(key, "ndim") and key.ndim > 1:
   1418         raise ValueError("Cannot index with multidimensional key")
-> 1420     return self._getitem_iterable(key, axis=axis)
   1422 # nested tuple slicing
   1423 if is_nested_tuple(key, labels):

File c:\Users\miles\.pyenv\pyenv-win\versions\3.12.5\Lib\site-packages\pandas\core\indexing.py:1360, in _LocIndexer._
getitem_iterable(self, key, axis)
   1357 self._validate_key(key, axis)
   1359 # A collection of keys
-> 1360 keyarr, indexer = self._get_listlike_indexer(key, axis)
   1361 return self.obj._reindex_with_indexers(
   1362     {axis: [keyarr, indexer]}, copy=True, allow_dups=True
   1363 )

File c:\Users\miles\.pyenv\pyenv-win\versions\3.12.5\Lib\site-packages\pandas\core\indexing.py:1558, in _LocIndexer._
get_listlike_indexer(self, key, axis)
   1555 ax = self.obj._get_axis(axis)
   1556 axis_name = self.obj._get_axis_name(axis)
-> 1558 keyarr, indexer = ax._get_indexer_strict(key, axis_name)
   1560 return keyarr, indexer

File c:\Users\miles\.pyenv\pyenv-win\versions\3.12.5\Lib\site-packages\pandas\core\indexes\base.py:6200, in Index._ge
t_indexer_strict(self, key, axis_name)
   6197 else:
   6198     keyarr, indexer, new_indexer = self._reindex_non_unique(keyarr)
-> 6200 self._raise_if_missing(keyarr, indexer, axis_name)
   6202 keyarr = self.take(indexer)
```

```
   6203 if isinstance(key, Index):
   6204     # GH 42790 - Preserve name from an Index

File c:\Users\miles\.pyenv\pyenv-win\versions\3.12.5\Lib\site-packages\pandas\core\indexes\base.py:6252, in Index._raise_if_missing(self, key, indexer, axis_name)
   6249     raise KeyError(f"None of [{key}] are in the [{axis_name}]")
   6251 not_found = list(ensure_index(key)[missing_mask.nonzero()[0]].unique())
-> 6252 raise KeyError(f"{not_found} not in index")

KeyError: "['e'] not in index"
```

Dropping rows or columns

you can use `df.drop()` to remove rows (default) or columns (specify axis = 1) at certain index locations.

In [53]:
```
df = pd.DataFrame(np.arange(12).reshape(3,4), columns=['A', 'B', 'C', 'D'], index = ['x','y','z'])
df
```

Out[53]:

	A	B	C	D
x	0	1	2	3
y	4	5	6	7
z	8	9	10	11

In [54]:
```
# drop rows
df.drop(['x', 'z'])
```

Out[54]:

	A	B	C	D
y	4	5	6	7

In [55]:
```
# drop columns
df.drop(['B', 'C'], axis = 1) # we must specify axis = 1 otherwise Pandas will look for "B" and "C" in the row names
```

Out[55]:

	A	D
x	0	3
y	4	7
z	8	11

In [56]:
```
# df.drop returns a new object and leaves df unchanged
# you can change this behavior with the argument inplace = True
df
```

Out[56]:

	A	B	C	D
x	0	1	2	3
y	4	5	6	7
z	8	9	10	11

Data Alignment

When performing element-wise arithmetic, Pandas will align the index values before doing the computation

In [57]:
```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s1
```

Out[57]:
```
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
```

In [58]:
```
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
               index=['a', 'c', 'e', 'f', 'g'])
s2
```

Out[58]:
```
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

In [59]:
```
pd.DataFrame({'s1':s1,'s2':s2}) # for reference
```

Out[59]:

	s1	s2
a	7.3	-2.1
c	-2.5	3.6
d	3.4	NaN
e	1.5	-1.5
f	NaN	4.0
g	NaN	3.1

In [60]:
```
s1 + s2  # returns a new series, where the indexes are the union of the indexes of s1 and s2
```

Out[60]:
```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

In [61]:
```
s1.add(s2)
```

Out[61]:
```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

In [62]:
```
pd.DataFrame({'s1':s1,'s2':s2})
```

Out[62]:

	s1	s2
a	7.3	-2.1
c	-2.5	3.6
d	3.4	NaN
e	1.5	-1.5
f	NaN	4.0
g	NaN	3.1

In [63]:
```
s1.sub(s2, fill_value = 0)
```

Out[63]:
```
a    9.4
c   -6.1
d    3.4
e    3.0
f   -4.0
g   -3.1
dtype: float64
```

In [64]:
```
s1.rsub(s2, fill_value = 0) # .rsub means 'right hand subtract' sets the series in the argument as the base
```

Out[64]:
```
a   -9.4
c    6.1
d   -3.4
e   -3.0
f    4.0
g    3.1
dtype: float64
```

In [65]:
```
s1 * s2
```

Out[65]:
```
a   -15.33
c    -9.00
d      NaN
e    -2.25
f      NaN
g      NaN
dtype: float64
```

In [66]:
```
s1.multiply(s2, fill_value = 1)
```

Out[66]:
```
a   -15.33
c    -9.00
d     3.40
e    -2.25
f     4.00
g     3.10
dtype: float64
```

For data frames with different columns, the rows and columns will be aligned

In [67]:
```
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
                   index=['Ohio', 'Texas', 'Colorado'])
df1
```

Out[67]:

	b	c	d
Ohio	0.0	1.0	2.0
Texas	3.0	4.0	5.0
Colorado	6.0	7.0	8.0

In [68]:
```
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
                   index=['Utah', 'Ohio', 'Texas', 'Oregon'])
df2
```

Out[68]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [69]:
```
df1 + df2
# c is in df1, but not df2
# e is in df2, but not df1
# the result returns the union of columns, but will fill in NaN for elements that do not exist in both
```

Out[69]:

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0	NaN	6.0	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0	NaN	12.0	NaN
Utah	NaN	NaN	NaN	NaN

In [70]:
```
# if you want to fill in values that are missing, you can use df.add() and specify the fill_value
# this will perform the above operation, but instead of using NaN when it can't find a value
# (which will return NaN),
# it will use the fill_value
df1.add(df2, fill_value = 0)
# you still get NaN if the value does not exist in either DataFrame
```

Out[70]:

	b	c	d	e
Colorado	6.0	7.0	8.0	NaN
Ohio	3.0	1.0	6.0	5.0
Oregon	9.0	NaN	10.0	11.0
Texas	9.0	4.0	12.0	8.0
Utah	0.0	NaN	1.0	2.0

Arithmetic operations that can be called on DataFrames and Series are:

- `.add()`, `.radd()` and `.sub()`, `.rsub()`
- `.mul()`, `.rmul()` and `.div()`, `.rdiv()`
- `.floordiv()`, `.rfloordiv()` (floor division `//`)
- `.pow()`, `.rpow()` (exponentiation `**`)

Summary Stats of a DataFrame

In [71]:
```
df = pd.DataFrame({'one':[1.5,6.0,np.nan, 1.5,4,6, np.nan],
                   'two':[np.nan, -4.5, np.nan, -1.5, 0, -4.5, 4]},
                  index=['a', 'b', 'c', 'd','e','f','g'])
df
```

Out[71]:

	one	two
a	1.5	NaN
b	6.0	-4.5
c	NaN	NaN
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	NaN	4.0

In [72]:
```
df.sum()  # default behavior returns column sums and skips missing values
# default behavior sums across axis 0 (sums the row)
```

Out[72]:
```
one    19.0
two    -6.5
dtype: float64
```

In [73]:
```
df # for reference
```

Out[73]:

	one	two
a	1.5	NaN
b	6.0	-4.5
c	NaN	NaN
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	NaN	4.0

In [74]:
```
df.sum(axis = 1) # sum across axis=1, sum across the columns and give row sums
```

Out[74]:
```
a    1.5
b    1.5
c    0.0
d    0.0
e    4.0
f    1.5
g    4.0
dtype: float64
```

In [75]:
```
df.sum(skipna = False)
```

Out[75]:
```
one   NaN
two   NaN
dtype: float64
```

In [76]:
```
df.mean()
```

Out[76]:
```
one    3.8
two   -1.3
dtype: float64
```

In [77]:
```
df.mean(axis = 1)
```

Out[77]:
```
a    1.50
b    0.75
c     NaN
d    0.00
e    2.00
f    0.75
g    4.00
dtype: float64
```

In [78]:
```
df # for reference
```

Out[78]:

	one	two
a	1.5	NaN
b	6.0	-4.5
c	NaN	NaN
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	NaN	4.0

In [79]:
```
df.min()
```

Out[79]:
```
one    1.5
two   -4.5
dtype: float64
```

In [80]:
```
df.idxmin()  # which row has the minimum value, also .idxmax()
# returns the first minimum, if there are multiple
# you can also specify axis
```

Out[80]:
```
one    a
two    b
dtype: object
```

Summary stats available for dataframes and series

- `count()` - number of non NA values
- `quantile()`
- `sum()`
- `mean()`
- `median()`
- `mad()` - mean absolute deviation
- `prod()`

- `var()`, `std()`

https://pandas.pydata.org/pandas-docs/stable/reference/series.html#computations-descriptive-stats

Unique values

In [81]:
```
df # for reference
```

Out[81]:

	one	two
a	1.5	NaN
b	6.0	-4.5
c	NaN	NaN
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	NaN	4.0

In [82]:
```
df.one.unique()  # shows the unique values in the order observed
```

Out[82]:
```
array([1.5, 6. , nan, 4. ])
```

In [83]:
```
df.two.unique()
```

Out[83]:
```
array([ nan, -4.5, -1.5,  0. ,  4. ])
```

In [84]:
```
df.unique()  # unique can only be applied to a series (a column in a dataframe)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_2740\1052518.py in ?()
----> 1 df.unique()  # unique can only be applied to a series (a column in a dataframe)

c:\Users\miles\.pyenv\pyenv-win\versions\3.12.5\Lib\site-packages\pandas\core\generic.py in ?(self, name)
   6295             and name not in self._accessors
   6296             and self._info_axis._can_hold_identifiers_and_holds_name(name)
   6297         ):
   6298             return self[name]
-> 6299         return object.__getattribute__(self, name)

AttributeError: 'DataFrame' object has no attribute 'unique'
```

In [85]:
```
df # for reference
```

Out[85]:

	one	two
a	1.5	NaN
b	6.0	-4.5
c	NaN	NaN
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	NaN	4.0

In [86]:
```
df.one.nunique()  # number of non-missing unique values exist
```

Out[86]: 3

In [87]:
```
df.one.value_counts()  # tally up counts of each value
# returns a series. the index are the unique values observed, the values are the frequencies.
# they appear in descending order of frequency
```

Out[87]:
```
one
1.5    2
6.0    2
4.0    1
Name: count, dtype: int64
```

In [88]:
```
df.one.isin([1.5, 4.0]) # checks to see if the value has membership in a particular list
# returns a series with boolean values
```

Out[88]:
```
a     True
b    False
c    False
d     True
e     True
f    False
g    False
Name: one, dtype: bool
```

In [89]:
```
(df.one == 1.5) | (df.one == 4.0)  # must use bitwise or. .isin() is much prefered
```

Out[89]:
```
a     True
b    False
c    False
d     True
e     True
f    False
g    False
Name: one, dtype: bool
```

In [90]:
```
df.loc[  df.one.isin([1.5,4.0]),  ]  # can filter rows based on the .isin() membership
```

Out[90]:

	one	two
a	1.5	NaN
d	1.5	-1.5
e	4.0	0.0

filtering out missing values

In [91]:
```
df
```

Out[91]:

	one	two
a	1.5	NaN
b	6.0	-4.5
c	NaN	NaN
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	NaN	4.0

In [92]:
```
df.dropna() # gets rid of any row that is not complete
```

Out[92]:

	one	two
b	6.0	-4.5
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5

In [93]:
```
df.dropna(how = 'all')  # only drops rows that are entirely NaN
```

Out[93]:

	one	two
a	1.5	NaN
b	6.0	-4.5
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	NaN	4.0

In [94]:
```
# you can also use .notnull(), which is True for values that are not missing
df[df.two.notnull()]  # You can use this in conjuntion with specifying the column
```

Out[94]:

	one	two
b	6.0	-4.5
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	NaN	4.0

Filling in Missing Values

In [95]:
```
df
```

Out[95]:

	one	two
a	1.5	NaN
b	6.0	-4.5
c	NaN	NaN
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	NaN	4.0

In [96]:
```
df.fillna(0) # fill in missing values with a constant
```

Out[96]:

	one	two
a	1.5	0.0
b	6.0	-4.5
c	0.0	0.0
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	0.0	4.0

In [97]:
```
df.fillna({'one': 1000, 'two': 0})  # use a dictionary to specify values to use for each column
```

Out[97]:

	one	two
a	1.5	0.0
b	6.0	-4.5
c	1000.0	0.0
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	1000.0	4.0

In [98]:
```
df.fillna(method = 'bfill')  # backfills. You can also use ffill
```

```
C:\Users\miles\AppData\Local\Temp\ipykernel_2740\233471607.py:1: FutureWarning: DataFrame.fillna with 'method' is dep
recated and will raise in a future version. Use obj.ffill() or obj.bfill() instead.
  df.fillna(method = 'bfill')  # backfills. You can also use ffill
```

Out[98]:

	one	two
a	1.5	-4.5
b	6.0	-4.5
c	1.5	-1.5
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	NaN	4.0

In [99]:
```
df.mean()
```

Out[99]:
```
one    3.8
two   -1.3
dtype: float64
```

In [100…
```
df.fillna(df.mean())  # fill na with df.mean() will fill in the column means
```

Out[100…

	one	two
a	1.5	-1.3
b	6.0	-4.5
c	3.8	-1.3
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	3.8	4.0

all of the above fillna methods have created new DataFrame objects. If you want to modify the current DataFrame, you can use the optional argument `inplace = True`

In [101…
```
df.T
```

Out[101…

	a	b	c	d	e	f	g
one	1.5	6.0	NaN	1.5	4.0	6.0	NaN
two	NaN	-4.5	NaN	-1.5	0.0	-4.5	4.0

In [102…
```
# apparently you can only fill missing values with dictionaries/series over a column
# so we have to do some Transpose magic
df.T.fillna(df.T.mean()).T
```

Out[102…

	one	two
a	1.5	1.5
b	6.0	-4.5
c	NaN	NaN
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	4.0	4.0

dealing with duplicates

In [103…

```
df
```

Out[103…

	one	two
a	1.5	NaN
b	6.0	-4.5
c	NaN	NaN
d	1.5	-1.5
e	4.0	0.0
f	6.0	-4.5
g	NaN	4.0

In [104…

```
df.duplicated()  # sees if any of the rows are a duplicate of an earlier row
```

Out[104…
```
a    False
b    False
c    False
d    False
e    False
f     True
g    False
dtype: bool
```

In [105…
```
df[~df.duplicated()]  # gets rid of the duplicated rows
```

Out[105…

	one	two
a	1.5	NaN
b	6.0	-4.5
c	NaN	NaN
d	1.5	-1.5
e	4.0	0.0
g	NaN	4.0

In [106…
```
df.one.duplicated()
```

Out[106…
```
a    False
b    False
c    False
d     True
e    False
f     True
g     True
Name: one, dtype: bool
```