Important Pandas Functions

Adapted from Wes McKinney's Python for Data Analysis and the Pandas Documentation

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

Redefining the Index

If you need to change the index of a series or dataframe, you can just define the index to something new.

In [2]:
```
original = pd.Series([1.4, 2.3, 3.1, 4.2], index = ['d','c','a','b'])
```

In [3]:
```
original
```

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

In [4]:
```
original['d':'a'] # can select values
```

Out[4]:
```
d    1.4
c    2.3
a    3.1
dtype: float64
```

In [5]:
```
original.index  # the original index are the letters d,c,a,b in a list
```

Out[5]:
```
Index(['d', 'c', 'a', 'b'], dtype='object')
```

In [6]:
```
original.index = range(4) # I assign the range object to be the index
```

In [7]:
```
original
```

Out[7]:
```
0    1.4
1    2.3
2    3.1
3    4.2
dtype: float64
```

In [8]:
```
original.index # We can see this has automatically become a RangeIndex object
```

Out[8]:
```
RangeIndex(start=0, stop=4, step=1)
```

In [9]:
```
original[1]
```

Out[9]:
```
2.3
```

In [10]:
```
original.loc[1] # behaves the same as above
```

Out[10]:
```
2.3
```

In [11]:
```
original.iloc[1] # behaves the same as above because the range index starts at 0
```

Out[11]:
```
2.3
```

In [12]:
```
original.index = range(1,5)
```

In [13]:
```
original
```

Out[13]:
```
1    1.4
2    2.3
3    3.1
4    4.2
dtype: float64
```

In [14]:
```
original[1]
```

Out[14]:
```
1.4
```

In [15]:
```
original.loc[1]
```

Out[15]:
```
1.4
```

In [16]:
```
original.iloc[1] # behavior is different because range index starts at 1
```

Out[16]:
```
2.3
```

In [17]:
```
original['a'] # throws an error because 'a' is no longer part of the index and cann
ot be used to select values
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-17-3915dcda73c1> in <module>
----> 1 original['a'] # throws an error because 'a' is no longer part of the i
ndex and cannot be used to select values

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\series.py in __getitem_
_(self, key)
    869         key = com.apply_if_callable(key, self)
    870         try:
--> 871             result = self.index.get_value(self, key)
    872
    873             if not is_scalar(result):

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_
value(self, series, key)
   4402         k = self._convert_scalar_indexer(k, kind="getitem")
   4403         try:
-> 4404             return self._engine.get_value(s, k, tz=getattr(series.dtyp
e, "tz", None))
   4405         except KeyError as e1:
   4406             if len(self) > 0 and (self.holds_integer() or self.is_bool
ean()):

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_value()

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_value()

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas\_libs\index_class_helper.pxi in pandas._libs.index.Int64Engine._check_t
ype()

KeyError: 'a'
```

In [18]:
```
original.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 [19]:
```
original
```

Out[19]:
```
a    1.4
b    2.3
c    3.1
d    4.2
dtype: float64
```

In [20]:
```
original['a']
```

Out[20]:
```
1.4
```

In [21]:
```
original[0]
```

Out[21]:
```
1.4
```

In [22]:
```
original.index = [1, 2, 3, 4, 5] # if the object you provide is of a different leng
th, you get a value error
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-22-36a203028421> in <module>
----> 1 original.index = [1, 2, 3, 4, 5] # if the object you provide is of a d
ifferent length, you get a value error

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\generic.py in __setattr
__(self, name, value)
   5285         try:
   5286             object.__getattribute__(self, name)
-> 5287             return object.__setattr__(self, name, value)
   5288         except AttributeError:
   5289             pass

pandas\_libs\properties.pyx in pandas._libs.properties.AxisProperty.__set__()

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\series.py in _set_axis
(self, axis, labels, fastpath)
    399         object.__setattr__(self, "_index", labels)
    400         if not fastpath:
--> 401             self._data.set_axis(axis, labels)
    402
    403     def _set_subtyp(self, is_all_dates):

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\internals\managers.py i
n set_axis(self, axis, new_labels)
    176         if new_len != old_len:
    177             raise ValueError(
--> 178                 f"Length mismatch: Expected axis has {old_len} element
s, new "
    179                 f"values have {new_len} elements"
    180             )

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

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 [23]:
```
original = pd.Series([1.4, 2.3, 3.1, 4.2], index = ['d','c','a','b'])
```

In [24]:
```
original
```

Out[24]:
```
d    1.4
c    2.3
a    3.1
b    4.2
dtype: float64
```

In [25]:
```
newobj = original.reindex(['a','b','c','d','e']) # note this has an index value tha
t doesn't exist in the original series
```

In [26]:
```
newobj  # takes the data in orignal and moves it so it conforms to the specified in
dex
# values that do not exist for the new index get NaN
```

Out[26]:
```
a    3.1
b    4.2
c    2.3
d    1.4
e    NaN
dtype: float64
```

In [27]:
```
# 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[27]:
```
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 [28]:
```
obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 3, 6])
obj3
```

Out[28]:
```
0      blue
3    purple
6    yellow
dtype: object
```

In [29]:
```
obj3.reindex(range(9))  # without any optional arguments, lots of missing values
```

Out[29]:
```
0      blue
1       NaN
2       NaN
3    purple
4       NaN
5       NaN
6    yellow
7       NaN
8       NaN
dtype: object
```

In [30]:
```
obj3.reindex(range(9), method='ffill')
# forward-fill pushes values 'forward' until a new value is encountered
```

Out[30]:
```
0      blue
1      blue
2      blue
3    purple
4    purple
5    purple
6    yellow
7    yellow
8    yellow
dtype: object
```

In [31]:
```
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[31]:
```
0      blue
1    purple
2    purple
3    purple
4    yellow
5    yellow
6    yellow
7       NaN
8       NaN
dtype: object
```

In [32]:
```
# 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[32]:
```
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 [33]:
```
# we create a DataFrame with the date index
df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]}, index=date_index)
df2
```

Out[33]:

	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 [34]:
```
date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')  # a new date index
df2.reindex(date_index2)
```

Out[34]:

	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 [35]:
```
df2.reindex(date_index2, method = 'bfill')
# he jan 3 isn't filled in because that NaN was not created by the reindexing proce
ss
# The NaN already existed in the data.
```

Out[35]:

	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 the same If the new index will have values that don't exist in the current index, you need to use reindex

In [36]:
```
obj5 = pd.DataFrame({'val':[1.4, 2.3, 3.1, 4.2]}, index = ['d','c','a','b'])
obj5
```

Out[36]:

	val
d	1.4
c	2.3
a	3.1
b	4.2

In [37]:
```
obj5.reindex(['a','b','c','d'])
```

Out[37]:

	val
a	3.1
b	4.2
c	2.3
d	1.4

In [38]:
```
obj5.loc[['a','b','c','d']] # works the same as reindex
```

Out[38]:

	val
a	3.1
b	4.2
c	2.3
d	1.4

In [39]:
```
obj5.reindex(['a','b','c','d','e'])
```

Out[39]:

	val
a	3.1
b	4.2
c	2.3
d	1.4
e	NaN

In [40]:
```
obj5.loc[['a','b','c','d','e']]  # .loc() returns a warning or error if you give an entry in the index that doesn't exist
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-40-b9b5ec5c39e9> in <module>
----> 1 obj5.loc[['a','b','c','d','e']]  # .loc() returns a warning or error i
f you give an entry in the index that doesn't exist

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexing.py in __getite
m__(self, key)
   1766
   1767             maybe_callable = com.apply_if_callable(key, self.obj)
-> 1768             return self._getitem_axis(maybe_callable, axis=axis)
   1769
   1770     def _is_scalar_access(self, key: Tuple):

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexing.py in _getitem
_axis(self, key, axis)
   1952                     raise ValueError("Cannot index with multidimension
al key")
   1953
-> 1954                 return self._getitem_iterable(key, axis=axis)
   1955
   1956             # nested tuple slicing

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexing.py in _getitem
_iterable(self, key, axis)
   1593         else:
   1594             # A collection of keys
-> 1595             keyarr, indexer = self._get_listlike_indexer(key, axis, ra
ise_missing=False)
   1596             return self.obj._reindex_with_indexers(
   1597                 {axis: [keyarr, indexer]}, copy=True, allow_dups=True

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexing.py in _get_lis
tlike_indexer(self, key, axis, raise_missing)
   1551
   1552         self._validate_read_indexer(
-> 1553             keyarr, indexer, o._get_axis_number(axis), raise_missing=r
```

In [41]:
```
obj5.loc[['a','b','c','d','a']]  # .loc() returns a warning or error if you give an entry in the index that doesn't exist
```

Out[41]:

	val
a	3.1
b	4.2
c	2.3
d	1.4
a	3.1

In [42]:
```
obj5 = obj5.reindex(['a','b','c','d','a'])
```

In [43]:
```
obj5
```

Out[43]:

	val
a	3.1
b	4.2
c	2.3
d	1.4
a	3.1

In [44]:
```
obj5.loc['c']
```

Out[44]:
```
val    2.3
Name: c, dtype: float64
```

In [45]:
```
obj5.loc['a'] = 5
```

In [46]:
```
obj5.loc['a']
```

Out[46]:

	val
a	5.0
a	5.0

In [47]:
```
obj5
```

Out[47]:

	val
a	5.0
b	4.2
c	2.3
d	1.4
a	5.0

Dropping rows or columns

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

In [48]:
```
df = pd.DataFrame(np.arange(12).reshape(3,4), columns=['A', 'B', 'C', 'D'], index =
['a','b','c'])
df
```

Out[48]:

	A	B	C	D
a	0	1	2	3
b	4	5	6	7
c	8	9	10	11

In [49]:
```
# drop rows
df.drop(['a', 'c'])
```

Out[49]:

	A	B	C	D
b	4	5	6	7

In [50]:
```
# drop columns
df.drop(['B', 'C'], axis=1)
```

Out[50]:

	A	D
a	0	3
b	4	7
c	8	11

In [51]:
```
# df.drop returns a new object and leaves df unchanged
# you can change this behavior with the argument inplace = True
df
```

Out[51]:

	A	B	C	D
a	0	1	2	3
b	4	5	6	7
c	8	9	10	11

Data Alignment

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

In [52]:
```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s1
```

Out[52]:
```
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
```

In [53]:
```
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
               index=['a', 'c', 'e', 'f', 'g'])
s2
```

Out[53]:
```
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

In [54]:
```
s1 + s2  # returns a new series, where the indexes are the union of the indexes of s1 and s2
```

Out[54]:
```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

In [55]:
```
s1.add(s2)
```

Out[55]:
```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

In [56]:
```
s1.add(s2, fill_value = 0)
```

Out[56]:
```
a    5.2
c    1.1
d    3.4
e    0.0
f    4.0
g    3.1
dtype: float64
```

In [57]:
```
s1 * s2
```

Out[57]:
```
a   -15.33
c    -9.00
d      NaN
e    -2.25
f      NaN
g      NaN
dtype: float64
```

In [58]:
```
s1.multiply(s2, fill_value = 1)
```

Out[58]:
```
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 [59]:
```
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
                   index=['Ohio', 'Texas', 'Colorado'])
df1
```

Out[59]:

	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 [60]:
```
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
                   index=['Utah', 'Ohio', 'Texas', 'Oregon'])
df2
```

Out[60]:

	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 [61]:
```
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 d
o not exist in both
```

Out[61]:

	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 [62]:
```
# 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 fin
d 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[62]:

	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

In [63]:
```
# other arithmetic operations that can be called on DataFrames are:
# .add()
# .sub()
# .mul()
# .div()
# .floordiv()
```

Summary Stats of a DataFrame

In [64]:
```
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[64]:

	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 [65]:
```
df.sum()  # default behavior returns column sums and skips missing values
# default behavior sums across axis 0 (sums the row)
```

Out[65]:
```
one    19.0
two    -6.5
dtype: float64
```

In [66]:
```
df.sum(axis = 1) # sum across axis=1, sum across the columns and give row sums
```

Out[66]:
```
a    1.5
b    1.5
c    0.0
d    0.0
e    4.0
f    1.5
g    4.0
dtype: float64
```

In [67]:
```
df.sum(skipna = False)
```

Out[67]:
```
one   NaN
two   NaN
dtype: float64
```

In [68]:
```
df.mean()
```

Out[68]:
```
one    3.8
two   -1.3
dtype: float64
```

In [69]:
```
df.mean(axis = 1)
```

Out[69]:
```
a    1.50
b    0.75
c     NaN
d    0.00
e    2.00
f    0.75
g    4.00
dtype: float64
```

In [70]:
```
df.min()
```

Out[70]:
```
one    1.5
two   -4.5
dtype: float64
```

In [71]:
```
df.idxmin()  # which row has the minimum value, also .idxmax()
# returns the first minimum, if there are multiple
```

Out[71]:
```
one    a
two    b
dtype: object
```

In [72]:
```
df.idxmax(axis = 1)
```

Out[72]:
```
a    one
b    one
c    NaN
d    one
e    one
f    one
g    two
dtype: object
```

In [73]:
```
df.one.unique()  # shows the unique values in the order observed
```

Out[73]:
```
array([1.5, 6. , nan, 4. ])
```

In [74]:
```
df.two.unique()
```

Out[74]:
```
array([ nan, -4.5, -1.5,  0. ,  4. ])
```

In [75]:
```
df.unique()  # unique can only be applied to a series (a column in a dataframe)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-75-02a393eeccfb> in <module>
----> 1 df.unique()  # unique can only be applied to a series (a column in a d
ataframe)

C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\generic.py in __getattr
__(self, name)
   5272             if self._info_axis._can_hold_identifiers_and_holds_name(na
me):
   5273                 return self[name]
-> 5274             return object.__getattribute__(self, name)
   5275
   5276     def __setattr__(self, name: str, value) -> None:

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

In [76]:
```
df.one.nunique()  # number of non-missing unique values exist
```

Out[76]:
```
3
```

In [77]:
```
df.one.value_counts()  # tally up counts of each value
# returns a series. the index are the unique values observed, the values are the fr
equencies.
# they appear in descending order of frequency
```

Out[77]:
```
6.0    2
1.5    2
4.0    1
Name: one, dtype: int64
```

In [78]:
```
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[78]:
```
a     True
b    False
c    False
d     True
e     True
f    False
g    False
Name: one, dtype: bool
```

In [79]:
```
(df.one == 1.5) | (df.one == 4.0)  # must use bitwise or. .isin() is much prefered
```

Out[79]:
```
a     True
b    False
c    False
d     True
e     True
f    False
g    False
Name: one, dtype: bool
```

In [80]:
```
df.loc[  df.one.isin([1.5,4.0]),  ]  # can filter rows based on the .isin() membership
```

Out[80]:

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

filtering out missing values

In [81]:
```
df
```

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.dropna() # gets rid of any row that is not complete
```

Out[82]:

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

In [83]:
```
df.dropna(how = 'all')   # only drops rows that are entirely NaN
```

Out[83]:

	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 [84]:
```
# 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[84]:

	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 [85]:
```
df
```

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.fillna(0) # fill in missing values with a constant
```

Out[86]:

	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 [87]:
```
df.fillna({'one': 1000, 'two': 0})  # use a dictionary to specify values to use for each column
```

Out[87]:

	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 [88]:
```
df.fillna(method = 'bfill')  # backfills. You can also use ffill
```

Out[88]:

	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 [89]:
```
df.mean()
```

Out[89]:
```
one     3.8
two    -1.3
dtype: float64
```

In [90]:
```
df.fillna(df.mean())  # fill na with df.mean() will fill in the column means
```

Out[90]:

	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 [91]:
```
df.T
```

Out[91]:

	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 [92]:
```
# apparently you can only fill missing values with dictionaries/series over a colum
n
# so we have to do some Transpose magic
df.T.fillna(df.T.mean()).T
```

Out[92]:

	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 [93]:
```
df
```

Out[93]:

	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 [94]:
```
df.duplicated()  # sees if any of the rows are a duplicate of an earlier row
```

Out[94]:
```
a    False
b    False
c    False
d    False
e    False
f     True
g    False
dtype: bool
```

In [95]:
```
df[~df.duplicated()]  # gets rid of the duplicated rows
```

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
g	NaN	4.0

In [96]:
```
df.one.duplicated()
```

Out[96]:
```
a    False
b    False
c    False
d     True
e    False
f     True
g     True
Name: one, dtype: bool
```