```
In [333]: dft = pd.DataFrame({'A': np.random.rand(3),
   .....:                     'B': 1,
   .....:                     'C': 'foo',
   .....:                     'D': pd.Timestamp('20010102'),
   .....:                     'E': pd.Series([1.0] * 3).astype('float32'),
   .....:                     'F': False,
   .....:                     'G': pd.Series([1] * 3, dtype='int8')})
   .....:

In [334]: dft
Out[334]:
          A  B    C          D    E      F  G
0  0.035962  1  foo 2001-01-02  1.0  False  1
1  0.701379  1  foo 2001-01-02  1.0  False  1
2  0.281885  1  foo 2001-01-02  1.0  False  1

In [335]: dft.dtypes
Out[335]:
A           float64
B             int64
C            object
D    datetime64[ns]
E           float32
F              bool
G              int8
dtype: object
```

On a `Series` object, use the *`dtype`* attribute.

```
In [336]: dft['A'].dtype
Out[336]: dtype('float64')
```

If a pandas object contains data with multiple dtypes *in a single column*, the dtype of the column will be chosen to accommodate all of the data types (`object` is the most general).

```
# these ints are coerced to floats
In [337]: pd.Series([1, 2, 3, 4, 5, 6.])
Out[337]:
0    1.0
1    2.0
2    3.0
3    4.0
4    5.0
5    6.0
dtype: float64

# string data forces an ``object`` dtype
In [338]: pd.Series([1, 2, 3, 6., 'foo'])
Out[338]:
0      1
1      2
2      3
3      6
4    foo
dtype: object
```

The number of columns of each type in a `DataFrame` can be found by calling `DataFrame.dtypes.value_counts()`.

```
In [339]: dft.dtypes.value_counts()
Out[339]:
int64             1
float32           1
datetime64[ns]    1
object            1
int8              1
float64           1
bool              1
dtype: int64
```

Numeric dtypes will propagate and can coexist in DataFrames. If a dtype is passed (either directly via the `dtype` keyword, a passed `ndarray`, or a passed `Series`), then it will be preserved in DataFrame operations. Furthermore, different numeric dtypes will **NOT** be combined. The following example will give you a taste.

```
In [340]: df1 = pd.DataFrame(np.random.randn(8, 1), columns=['A'], dtype='float32')

In [341]: df1
Out[341]:
          A
0  0.224364
1  1.890546
2  0.182879
3  0.787847
4 -0.188449
5  0.667715
6 -0.011736
7 -0.399073

In [342]: df1.dtypes
Out[342]:
A    float32
dtype: object

In [343]: df2 = pd.DataFrame({'A': pd.Series(np.random.randn(8), dtype='float16'),
   .....:                     'B': pd.Series(np.random.randn(8)),
   .....:                     'C': pd.Series(np.array(np.random.randn(8),
   .....:                                             dtype='uint8'))})
   .....:

In [344]: df2
Out[344]:
          A         B    C
0  0.823242  0.256090    0
1  1.607422  1.426469    0
2 -0.333740 -0.416203  255
3 -0.063477  1.139976    0
4 -1.014648 -1.193477    0
5  0.678711  0.096706    0
6 -0.040863 -1.956850    1
7 -0.357422 -0.714337    0

In [345]: df2.dtypes
Out[345]:
A    float16
B    float64
C      uint8
```

(continues on next page)

(continued from previous page)

```
dtype: object
```

defaults

By default integer types are `int64` and float types are `float64`, *regardless* of platform (32-bit or 64-bit). The following will all result in `int64` dtypes.

```
In [346]: pd.DataFrame([1, 2], columns=['a']).dtypes
Out[346]:
a    int64
dtype: object

In [347]: pd.DataFrame({'a': [1, 2]}).dtypes
Out[347]:
a    int64
dtype: object

In [348]: pd.DataFrame({'a': 1}, index=list(range(2))).dtypes
Out[348]:
a    int64
dtype: object
```

Note that Numpy will choose *platform-dependent* types when creating arrays. The following **WILL** result in `int32` on 32-bit platform.

```
In [349]: frame = pd.DataFrame(np.array([1, 2]))
```

upcasting

Types can potentially be *upcasted* when combined with other types, meaning they are promoted from the current type (e.g. `int` to `float`).

```
In [350]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2

In [351]: df3
Out[351]:
          A         B      C
0  1.047606  0.256090    0.0
1  3.497968  1.426469    0.0
2 -0.150862 -0.416203  255.0
3  0.724370  1.139976    0.0
4 -1.203098 -1.193477    0.0
5  1.346426  0.096706    0.0
6 -0.052599 -1.956850    1.0
7 -0.756495 -0.714337    0.0

In [352]: df3.dtypes
Out[352]:
A    float32
B    float64
C    float64
dtype: object
```

DataFrame.to_numpy() will return the *lower-common-denominator* of the dtypes, meaning the dtype that can accommodate **ALL** of the types in the resulting homogeneous dtyped NumPy array. This can force some *upcasting*.

```
In [353]: df3.to_numpy().dtype
Out[353]: dtype('float64')
```

astype

You can use the *astype()* method to explicitly convert dtypes from one to another. These will by default return a copy, even if the dtype was unchanged (pass `copy=False` to change this behavior). In addition, they will raise an exception if the astype operation is invalid.

Upcasting is always according to the **numpy** rules. If two different dtypes are involved in an operation, then the more *general* one will be used as the result of the operation.

```
In [354]: df3
Out[354]:
          A         B      C
0  1.047606  0.256090    0.0
1  3.497968  1.426469    0.0
2 -0.150862 -0.416203  255.0
3  0.724370  1.139976    0.0
4 -1.203098 -1.193477    0.0
5  1.346426  0.096706    0.0
6 -0.052599 -1.956850    1.0
7 -0.756495 -0.714337    0.0

In [355]: df3.dtypes
Out[355]:
A    float32
B    float64
C    float64
dtype: object

# conversion of dtypes
In [356]: df3.astype('float32').dtypes
Out[356]:
A    float32
B    float32
C    float32
dtype: object
```

Convert a subset of columns to a specified type using *astype()*.

```
In [357]: dft = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})

In [358]: dft[['a', 'b']] = dft[['a', 'b']].astype(np.uint8)

In [359]: dft
Out[359]:
   a  b  c
0  1  4  7
1  2  5  8
2  3  6  9

In [360]: dft.dtypes
```

(continues on next page)

(continued from previous page)

```
Out[360]:
a    uint8
b    uint8
c    int64
dtype: object
```

Convert certain columns to a specific dtype by passing a dict to *astype()*.

```
In [361]: dft1 = pd.DataFrame({'a': [1, 0, 1], 'b': [4, 5, 6], 'c': [7, 8, 9]})

In [362]: dft1 = dft1.astype({'a': np.bool, 'c': np.float64})

In [363]: dft1
Out[363]:
       a  b    c
0   True  4  7.0
1  False  5  8.0
2   True  6  9.0

In [364]: dft1.dtypes
Out[364]:
a       bool
b      int64
c    float64
dtype: object
```

Note: When trying to convert a subset of columns to a specified type using *astype()* and *loc()*, upcasting occurs.

loc() tries to fit in what we are assigning to the current dtypes, while `[]` will overwrite them taking the dtype from the right hand side. Therefore the following piece of code produces the unintended result.

```
In [365]: dft = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})

In [366]: dft.loc[:, ['a', 'b']].astype(np.uint8).dtypes
Out[366]:
a    uint8
b    uint8
dtype: object

In [367]: dft.loc[:, ['a', 'b']] = dft.loc[:, ['a', 'b']].astype(np.uint8)

In [368]: dft.dtypes
Out[368]:
a    int64
b    int64
c    int64
dtype: object
```

object conversion

pandas offers various functions to try to force conversion of types from the `object` dtype to other types. In cases where the data is already of the correct type, but stored in an `object` array, the *DataFrame.infer_objects()* and *Series.infer_objects()* methods can be used to soft convert to the correct type.

```
In [369]: import datetime

In [370]: df = pd.DataFrame([[1, 2],
   .....:                    ['a', 'b'],
   .....:                    [datetime.datetime(2016, 3, 2),
   .....:                     datetime.datetime(2016, 3, 2)]])
   .....:

In [371]: df = df.T

In [372]: df
Out[372]:
   0  1          2
0  1  a 2016-03-02
1  2  b 2016-03-02

In [373]: df.dtypes
Out[373]:
0            object
1            object
2    datetime64[ns]
dtype: object
```

Because the data was transposed the original inference stored all columns as object, which `infer_objects` will correct.

```
In [374]: df.infer_objects().dtypes
Out[374]:
0             int64
1            object
2    datetime64[ns]
dtype: object
```

The following functions are available for one dimensional object arrays or scalars to perform hard conversion of objects to a specified type:

- *to_numeric()* (conversion to numeric dtypes)

```
In [375]: m = ['1.1', 2, 3]

In [376]: pd.to_numeric(m)
Out[376]: array([1.1, 2. , 3. ])
```

- *to_datetime()* (conversion to datetime objects)

```
In [377]: import datetime

In [378]: m = ['2016-07-09', datetime.datetime(2016, 3, 2)]

In [379]: pd.to_datetime(m)
Out[379]: DatetimeIndex(['2016-07-09', '2016-03-02'], dtype='datetime64[ns]',␣
↪freq=None)
```

- *to_timedelta()* (conversion to timedelta objects)

```
In [380]: m = ['5us', pd.Timedelta('1day')]

In [381]: pd.to_timedelta(m)
Out[381]: TimedeltaIndex(['0 days 00:00:00.000005', '1 days 00:00:00'], dtype=
↪'timedelta64[ns]', freq=None)
```

To force a conversion, we can pass in an `errors` argument, which specifies how pandas should deal with elements that cannot be converted to desired dtype or object. By default, `errors='raise'`, meaning that any errors encountered will be raised during the conversion process. However, if `errors='coerce'`, these errors will be ignored and pandas will convert problematic elements to `pd.NaT` (for datetime and timedelta) or `np.nan` (for numeric). This might be useful if you are reading in data which is mostly of the desired dtype (e.g. numeric, datetime), but occasionally has non-conforming elements intermixed that you want to represent as missing:

```
In [382]: import datetime

In [383]: m = ['apple', datetime.datetime(2016, 3, 2)]

In [384]: pd.to_datetime(m, errors='coerce')
Out[384]: DatetimeIndex(['NaT', '2016-03-02'], dtype='datetime64[ns]', freq=None)

In [385]: m = ['apple', 2, 3]

In [386]: pd.to_numeric(m, errors='coerce')
Out[386]: array([nan,  2.,  3.])

In [387]: m = ['apple', pd.Timedelta('1day')]

In [388]: pd.to_timedelta(m, errors='coerce')
Out[388]: TimedeltaIndex([NaT, '1 days'], dtype='timedelta64[ns]', freq=None)
```

The `errors` parameter has a third option of `errors='ignore'`, which will simply return the passed in data if it encounters any errors with the conversion to a desired data type:

```
In [389]: import datetime

In [390]: m = ['apple', datetime.datetime(2016, 3, 2)]

In [391]: pd.to_datetime(m, errors='ignore')
Out[391]: Index(['apple', 2016-03-02 00:00:00], dtype='object')

In [392]: m = ['apple', 2, 3]

In [393]: pd.to_numeric(m, errors='ignore')
Out[393]: array(['apple', 2, 3], dtype=object)

In [394]: m = ['apple', pd.Timedelta('1day')]

In [395]: pd.to_timedelta(m, errors='ignore')
Out[395]: array(['apple', Timedelta('1 days 00:00:00')], dtype=object)
```

In addition to object conversion, *to_numeric()* provides another argument `downcast`, which gives the option of downcasting the newly (or already) numeric data to a smaller dtype, which can conserve memory:

```
In [396]: m = ['1', 2, 3]
```

(continues on next page)

(continued from previous page)

```
In [397]: pd.to_numeric(m, downcast='integer')    # smallest signed int dtype
Out[397]: array([1, 2, 3], dtype=int8)

In [398]: pd.to_numeric(m, downcast='signed')     # same as 'integer'
Out[398]: array([1, 2, 3], dtype=int8)

In [399]: pd.to_numeric(m, downcast='unsigned')   # smallest unsigned int dtype
Out[399]: array([1, 2, 3], dtype=uint8)

In [400]: pd.to_numeric(m, downcast='float')      # smallest float dtype
Out[400]: array([1., 2., 3.], dtype=float32)
```

As these methods apply only to one-dimensional arrays, lists or scalars; they cannot be used directly on multi-dimensional objects such as DataFrames. However, with *apply()*, we can "apply" the function over each column efficiently:

```
In [401]: import datetime

In [402]: df = pd.DataFrame([
   .....:     ['2016-07-09', datetime.datetime(2016, 3, 2)]] * 2, dtype='O')
   .....:

In [403]: df
Out[403]:
            0                    1
0  2016-07-09  2016-03-02 00:00:00
1  2016-07-09  2016-03-02 00:00:00

In [404]: df.apply(pd.to_datetime)
Out[404]:
           0          1
0 2016-07-09 2016-03-02
1 2016-07-09 2016-03-02

In [405]: df = pd.DataFrame([['1.1', 2, 3]] * 2, dtype='O')

In [406]: df
Out[406]:
     0  1  2
0  1.1  2  3
1  1.1  2  3

In [407]: df.apply(pd.to_numeric)
Out[407]:
     0  1  2
0  1.1  2  3
1  1.1  2  3

In [408]: df = pd.DataFrame([['5us', pd.Timedelta('1day')]] * 2, dtype='O')

In [409]: df
Out[409]:
     0                1
0  5us  1 days 00:00:00
1  5us  1 days 00:00:00
```

(continues on next page)

(continued from previous page)

```
In [410]: df.apply(pd.to_timedelta)
Out[410]:
                 0      1
0 00:00:00.000005 1 days
1 00:00:00.000005 1 days
```

gotchas

Performing selection operations on `integer` type data can easily upcast the data to `floating`. The dtype of the input data will be preserved in cases where `nans` are not introduced. See also *Support for integer NA*.

```
In [411]: dfi = df3.astype('int32')

In [412]: dfi['E'] = 1

In [413]: dfi
Out[413]:
   A  B    C  E
0  1  0    0  1
1  3  1    0  1
2  0  0  255  1
3  0  1    0  1
4 -1 -1    0  1
5  1  0    0  1
6  0 -1    1  1
7  0  0    0  1

In [414]: dfi.dtypes
Out[414]:
A    int32
B    int32
C    int32
E    int64
dtype: object

In [415]: casted = dfi[dfi > 0]

In [416]: casted
Out[416]:
     A    B      C  E
0  1.0  NaN    NaN  1
1  3.0  1.0    NaN  1
2  NaN  NaN  255.0  1
3  NaN  1.0    NaN  1
4  NaN  NaN    NaN  1
5  1.0  NaN    NaN  1
6  NaN  NaN    1.0  1
7  NaN  NaN    NaN  1

In [417]: casted.dtypes
Out[417]:
A    float64
B    float64
C    float64
E      int64
```

(continues on next page)

(continued from previous page)

```
dtype: object
```

While float dtypes are unchanged.

```
In [418]: dfa = df3.copy()

In [419]: dfa['A'] = dfa['A'].astype('float32')

In [420]: dfa.dtypes
Out[420]:
A    float32
B    float64
C    float64
dtype: object

In [421]: casted = dfa[df2 > 0]

In [422]: casted
Out[422]:
          A         B      C
0  1.047606  0.256090    NaN
1  3.497968  1.426469    NaN
2       NaN       NaN  255.0
3       NaN  1.139976    NaN
4       NaN       NaN    NaN
5  1.346426  0.096706    NaN
6       NaN       NaN    1.0
7       NaN       NaN    NaN

In [423]: casted.dtypes
Out[423]:
A    float32
B    float64
C    float64
dtype: object
```

Selecting columns based on `dtype`

The *`select_dtypes()`* method implements subsetting of columns based on their `dtype`.

First, let's create a *`DataFrame`* with a slew of different dtypes:

```
In [424]: df = pd.DataFrame({'string': list('abc'),
   .....:                    'int64': list(range(1, 4)),
   .....:                    'uint8': np.arange(3, 6).astype('u1'),
   .....:                    'float64': np.arange(4.0, 7.0),
   .....:                    'bool1': [True, False, True],
   .....:                    'bool2': [False, True, False],
   .....:                    'dates': pd.date_range('now', periods=3),
   .....:                    'category': pd.Series(list("ABC")).astype('category')})
   .....:

In [425]: df['tdeltas'] = df.dates.diff()

In [426]: df['uint64'] = np.arange(3, 6).astype('u8')
```

(continues on next page)

(continued from previous page)

```
In [427]: df['other_dates'] = pd.date_range('20130101', periods=3)

In [428]: df['tz_aware_dates'] = pd.date_range('20130101', periods=3, tz='US/Eastern')

In [429]: df
Out[429]:
  string  int64  uint8  float64  bool1  bool2                     dates category␣
↪tdeltas  uint64 other_dates           tz_aware_dates
0      a      1      3      4.0   True  False 2020-06-17 17:43:44.044203        A␣
↪NaT       3  2013-01-01 2013-01-01 00:00:00-05:00
1      b      2      4      5.0  False   True 2020-06-18 17:43:44.044203        B  1␣
↪days      4  2013-01-02 2013-01-02 00:00:00-05:00
2      c      3      5      6.0   True  False 2020-06-19 17:43:44.044203        C  1␣
↪days      5  2013-01-03 2013-01-03 00:00:00-05:00
```

And the dtypes:

```
In [430]: df.dtypes
Out[430]:
string                                object
int64                                  int64
uint8                                  uint8
float64                              float64
bool1                                   bool
bool2                                   bool
dates                         datetime64[ns]
category                            category
tdeltas                     timedelta64[ns]
uint64                               uint64
other_dates                   datetime64[ns]
tz_aware_dates    datetime64[ns, US/Eastern]
dtype: object
```

select_dtypes() has two parameters `include` and `exclude` that allow you to say "give me the columns *with* these dtypes" (`include`) and/or "give the columns *without* these dtypes" (`exclude`).

For example, to select `bool` columns:

```
In [431]: df.select_dtypes(include=[bool])
Out[431]:
   bool1  bool2
0   True  False
1  False   True
2   True  False
```

You can also pass the name of a dtype in the NumPy dtype hierarchy:

```
In [432]: df.select_dtypes(include=['bool'])
Out[432]:
   bool1  bool2
0   True  False
1  False   True
2   True  False
```

select_dtypes() also works with generic dtypes as well.

For example, to select all numeric and boolean columns while excluding unsigned integers:

```
In [433]: df.select_dtypes(include=['number', 'bool'], exclude=['unsignedinteger'])
Out[433]:
   int64  float64  bool1  bool2 tdeltas
0      1      4.0   True  False     NaT
1      2      5.0  False   True  1 days
2      3      6.0   True  False  1 days
```

To select string columns you must use the `object` dtype:

```
In [434]: df.select_dtypes(include=['object'])
Out[434]:
  string
0      a
1      b
2      c
```

To see all the child dtypes of a generic `dtype` like `numpy.number` you can define a function that returns a tree of child dtypes:

```
In [435]: def subdtypes(dtype):
   .....:     subs = dtype.__subclasses__()
   .....:     if not subs:
   .....:         return dtype
   .....:     return [dtype, [subdtypes(dt) for dt in subs]]
   .....:
```

All NumPy dtypes are subclasses of `numpy.generic`:

```
In [436]: subdtypes(np.generic)
Out[436]:
[numpy.generic,
 [[numpy.number,
   [[numpy.integer,
     [[numpy.signedinteger,
       [numpy.int8,
        numpy.int16,
        numpy.int32,
        numpy.int64,
        numpy.longlong,
        numpy.timedelta64]],
      [numpy.unsignedinteger,
       [numpy.uint8,
        numpy.uint16,
        numpy.uint32,
        numpy.uint64,
        numpy.ulonglong]]]],
    [numpy.inexact,
     [[numpy.floating,
       [numpy.float16, numpy.float32, numpy.float64, numpy.float128]],
      [numpy.complexfloating,
       [numpy.complex64, numpy.complex128, numpy.complex256]]]]]],
  [numpy.flexible,
   [[numpy.character, [numpy.bytes_, numpy.str_]],
    [numpy.void, [numpy.record]]]],
  numpy.bool_,
  numpy.datetime64,
  numpy.object_]]
```

Note: Pandas also defines the types `category`, and `datetime64[ns, tz]`, which are not integrated into the normal NumPy hierarchy and won't show up with the above function.

1.4.6 Intro to data structures

We'll start with a quick, non-comprehensive overview of the fundamental data structures in pandas to get you started. The fundamental behavior about data types, indexing, and axis labeling / alignment apply across all of the objects. To get started, import NumPy and load pandas into your namespace:

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

In [2]: import pandas as pd
```

Here is a basic tenet to keep in mind: **data alignment is intrinsic**. The link between labels and data will not be broken unless done so explicitly by you.

We'll give a brief intro to the data structures, then consider all of the broad categories of functionality and methods in separate sections.

Series

Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the **index**. The basic method to create a Series is to call:

```
>>> s = pd.Series(data, index=index)
```

Here, `data` can be many different things:

- a Python dict
- an ndarray
- a scalar value (like 5)

The passed **index** is a list of axis labels. Thus, this separates into a few cases depending on what **data is**:

From ndarray

If `data` is an ndarray, **index** must be the same length as **data**. If no index is passed, one will be created having values `[0, ..., len(data) - 1]`.

```
In [3]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [4]: s
Out[4]:
a    0.469112
b   -0.282863
c   -1.509059
d   -1.135632
e    1.212112
dtype: float64

In [5]: s.index
Out[5]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

(continues on next page)

(continued from previous page)

```
In [6]: pd.Series(np.random.randn(5))
Out[6]:
0   -0.173215
1    0.119209
2   -1.044236
3   -0.861849
4   -2.104569
dtype: float64
```

Note: pandas supports non-unique index values. If an operation that does not support duplicate index values is attempted, an exception will be raised at that time. The reason for being lazy is nearly all performance-based (there are many instances in computations, like parts of GroupBy, where the index is not used).

From dict

Series can be instantiated from dicts:

```
In [7]: d = {'b': 1, 'a': 0, 'c': 2}

In [8]: pd.Series(d)
Out[8]:
b    1
a    0
c    2
dtype: int64
```

Note: When the data is a dict, and an index is not passed, the `Series` index will be ordered by the dict's insertion order, if you're using Python version >= 3.6 and Pandas version >= 0.23.

If you're using Python < 3.6 or Pandas < 0.23, and an index is not passed, the `Series` index will be the lexically ordered list of dict keys.

In the example above, if you were on a Python version lower than 3.6 or a Pandas version lower than 0.23, the `Series` would be ordered by the lexical order of the dict keys (i.e. `['a', 'b', 'c']` rather than `['b', 'a', 'c']`).

If an index is passed, the values in data corresponding to the labels in the index will be pulled out.

```
In [9]: d = {'a': 0., 'b': 1., 'c': 2.}

In [10]: pd.Series(d)
Out[10]:
a    0.0
b    1.0
c    2.0
dtype: float64

In [11]: pd.Series(d, index=['b', 'c', 'd', 'a'])
Out[11]:
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

Note: NaN (not a number) is the standard missing data marker used in pandas.

From scalar value

If `data` is a scalar value, an index must be provided. The value will be repeated to match the length of **index**.

```
In [12]: pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
Out[12]:
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

Series is ndarray-like

`Series` acts very similarly to a `ndarray`, and is a valid argument to most NumPy functions. However, operations such as slicing will also slice the index.

```
In [13]: s[0]
Out[13]: 0.4691122999071863

In [14]: s[:3]
Out[14]:
a    0.469112
b   -0.282863
c   -1.509059
dtype: float64

In [15]: s[s > s.median()]
Out[15]:
a    0.469112
e    1.212112
dtype: float64

In [16]: s[[4, 3, 1]]
Out[16]:
e    1.212112
d   -1.135632
b   -0.282863
dtype: float64

In [17]: np.exp(s)
Out[17]:
a    1.598575
b    0.753623
c    0.221118
d    0.321219
e    3.360575
dtype: float64
```

Note: We will address array-based indexing like `s[[4, 3, 1]]` in *section*.

Like a NumPy array, a pandas Series has a *dtype*.

```
In [18]: s.dtype
Out[18]: dtype('float64')
```

This is often a NumPy dtype. However, pandas and 3rd-party libraries extend NumPy's type system in a few places, in which case the dtype would be a *ExtensionDtype*. Some examples within pandas are *Categorical data* and *Nullable integer data type*. See *dtypes* for more.

If you need the actual array backing a `Series`, use *Series.array*.

```
In [19]: s.array
Out[19]:
<PandasArray>
[ 0.4691122999071863, -0.2828633443286633, -1.5090585031735124,
 -1.1356323710171934,  1.2121120250208506]
Length: 5, dtype: float64
```

Accessing the array can be useful when you need to do some operation without the index (to disable *automatic alignment*, for example).

Series.array will always be an *ExtensionArray*. Briefly, an ExtensionArray is a thin wrapper around one or more *concrete* arrays like a `numpy.ndarray`. Pandas knows how to take an `ExtensionArray` and store it in a `Series` or a column of a `DataFrame`. See *dtypes* for more.

While Series is ndarray-like, if you need an *actual* ndarray, then use *Series.to_numpy()*.

```
In [20]: s.to_numpy()
Out[20]: array([ 0.4691, -0.2829, -1.5091, -1.1356,  1.2121])
```

Even if the Series is backed by a *ExtensionArray*, *Series.to_numpy()* will return a NumPy ndarray.

Series is dict-like

A Series is like a fixed-size dict in that you can get and set values by index label:

```
In [21]: s['a']
Out[21]: 0.4691122999071863

In [22]: s['e'] = 12.

In [23]: s
Out[23]:
a     0.469112
b    -0.282863
c    -1.509059
d    -1.135632
e    12.000000
dtype: float64

In [24]: 'e' in s
Out[24]: True

In [25]: 'f' in s
Out[25]: False
```

If a label is not contained, an exception is raised:

```
>>> s['f']
KeyError: 'f'
```

Using the `get` method, a missing label will return None or specified default:

```
In [26]: s.get('f')

In [27]: s.get('f', np.nan)
Out[27]: nan
```

See also the *section on attribute access*.

Vectorized operations and label alignment with Series

When working with raw NumPy arrays, looping through value-by-value is usually not necessary. The same is true when working with Series in pandas. Series can also be passed into most NumPy methods expecting an ndarray.

```
In [28]: s + s
Out[28]:
a     0.938225
b    -0.565727
c    -3.018117
d    -2.271265
e    24.000000
dtype: float64

In [29]: s * 2
Out[29]:
a     0.938225
b    -0.565727
c    -3.018117
d    -2.271265
e    24.000000
dtype: float64

In [30]: np.exp(s)
Out[30]:
a         1.598575
b         0.753623
c         0.221118
d         0.321219
e    162754.791419
dtype: float64
```

A key difference between Series and ndarray is that operations between Series automatically align the data based on label. Thus, you can write computations without giving consideration to whether the Series involved have the same labels.

```
In [31]: s[1:] + s[:-1]
Out[31]:
a         NaN
b   -0.565727
c   -3.018117
d   -2.271265
e         NaN
dtype: float64
```

The result of an operation between unaligned Series will have the **union** of the indexes involved. If a label is not found in one Series or the other, the result will be marked as missing `NaN`. Being able to write code without doing any explicit data alignment grants immense freedom and flexibility in interactive data analysis and research. The integrated data alignment features of the pandas data structures set pandas apart from the majority of related tools for working with labeled data.

Note: In general, we chose to make the default result of operations between differently indexed objects yield the **union** of the indexes in order to avoid loss of information. Having an index label, though the data is missing, is typically important information as part of a computation. You of course have the option of dropping labels with missing data via the **dropna** function.

Name attribute

Series can also have a `name` attribute:

```
In [32]: s = pd.Series(np.random.randn(5), name='something')

In [33]: s
Out[33]:
0   -0.494929
1    1.071804
2    0.721555
3   -0.706771
4   -1.039575
Name: something, dtype: float64

In [34]: s.name
Out[34]: 'something'
```

The Series `name` will be assigned automatically in many cases, in particular when taking 1D slices of DataFrame as you will see below.

You can rename a Series with the *pandas.Series.rename()* method.

```
In [35]: s2 = s.rename("different")

In [36]: s2.name
Out[36]: 'different'
```

Note that `s` and `s2` refer to different objects.

DataFrame

DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object. Like Series, DataFrame accepts many different kinds of input:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Structured or record ndarray
- A `Series`
- Another `DataFrame`

Along with the data, you can optionally pass **index** (row labels) and **columns** (column labels) arguments. If you pass an index and / or columns, you are guaranteeing the index and / or columns of the resulting DataFrame. Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.

If axis labels are not passed, they will be constructed from the input data based on common sense rules.

Note: When the data is a dict, and `columns` is not specified, the `DataFrame` columns will be ordered by the dict's insertion order, if you are using Python version >= 3.6 and Pandas >= 0.23.

If you are using Python < 3.6 or Pandas < 0.23, and `columns` is not specified, the `DataFrame` columns will be the lexically ordered list of dict keys.

From dict of Series or dicts

The resulting **index** will be the **union** of the indexes of the various Series. If there are any nested dicts, these will first be converted to Series. If no columns are passed, the columns will be the ordered list of dict keys.

```
In [37]: d = {'one': pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
   ....:      'two': pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
   ....:

In [38]: df = pd.DataFrame(d)

In [39]: df
Out[39]:
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0

In [40]: pd.DataFrame(d, index=['d', 'b', 'a'])
Out[40]:
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0

In [41]: pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
Out[41]:
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
```

The row and column labels can be accessed respectively by accessing the **index** and **columns** attributes:

Note: When a particular set of columns is passed along with a dict of data, the passed columns override the keys in the dict.

```
In [42]: df.index
Out[42]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

(continues on next page)

(continued from previous page)

```
In [43]: df.columns
Out[43]: Index(['one', 'two'], dtype='object')
```

From dict of ndarrays / lists

The ndarrays must all be the same length. If an index is passed, it must clearly also be the same length as the arrays. If no index is passed, the result will be `range(n)`, where `n` is the array length.

```
In [44]: d = {'one': [1., 2., 3., 4.],
   ....:      'two': [4., 3., 2., 1.]}
   ....:

In [45]: pd.DataFrame(d)
Out[45]:
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0

In [46]: pd.DataFrame(d, index=['a', 'b', 'c', 'd'])
Out[46]:
   one  two
a  1.0  4.0
b  2.0  3.0
c  3.0  2.0
d  4.0  1.0
```

From structured or record array

This case is handled identically to a dict of arrays.

```
In [47]: data = np.zeros((2, ), dtype=[('A', 'i4'), ('B', 'f4'), ('C', 'a10')])

In [48]: data[:] = [(1, 2., 'Hello'), (2, 3., "World")]

In [49]: pd.DataFrame(data)
Out[49]:
   A    B         C
0  1  2.0  b'Hello'
1  2  3.0  b'World'

In [50]: pd.DataFrame(data, index=['first', 'second'])
Out[50]:
        A    B         C
first   1  2.0  b'Hello'
second  2  3.0  b'World'

In [51]: pd.DataFrame(data, columns=['C', 'A', 'B'])
Out[51]:
           C  A    B
0  b'Hello'  1  2.0
1  b'World'  2  3.0
```

Note: DataFrame is not intended to work exactly like a 2-dimensional NumPy ndarray.

From a list of dicts

```
In [52]: data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

In [53]: pd.DataFrame(data2)
Out[53]:
   a   b     c
0  1   2   NaN
1  5  10  20.0

In [54]: pd.DataFrame(data2, index=['first', 'second'])
Out[54]:
        a   b     c
first   1   2   NaN
second  5  10  20.0

In [55]: pd.DataFrame(data2, columns=['a', 'b'])
Out[55]:
   a   b
0  1   2
1  5  10
```

From a dict of tuples

You can automatically create a MultiIndexed frame by passing a tuples dictionary.

```
In [56]: pd.DataFrame({('a', 'b'): {('A', 'B'): 1, ('A', 'C'): 2},
   ....:                ('a', 'a'): {('A', 'C'): 3, ('A', 'B'): 4},
   ....:                ('a', 'c'): {('A', 'B'): 5, ('A', 'C'): 6},
   ....:                ('b', 'a'): {('A', 'C'): 7, ('A', 'B'): 8},
   ....:                ('b', 'b'): {('A', 'D'): 9, ('A', 'B'): 10}})
   ....:
Out[56]:
       a              b
       b    a    c    a     b
A B  1.0  4.0  5.0  8.0  10.0
  C  2.0  3.0  6.0  7.0   NaN
  D  NaN  NaN  NaN  NaN   9.0
```

From a Series

The result will be a DataFrame with the same index as the input Series, and with one column whose name is the original name of the Series (only if no other column name provided).

Missing data

Much more will be said on this topic in the *Missing data* section. To construct a DataFrame with missing data, we use `np.nan` to represent missing values. Alternatively, you may pass a `numpy.MaskedArray` as the data argument to the DataFrame constructor, and its masked entries will be considered missing.

Alternate constructors

DataFrame.from_dict

`DataFrame.from_dict` takes a dict of dicts or a dict of array-like sequences and returns a DataFrame. It operates like the `DataFrame` constructor except for the `orient` parameter which is `'columns'` by default, but which can be set to `'index'` in order to use the dict keys as row labels.

```
In [57]: pd.DataFrame.from_dict(dict([('A', [1, 2, 3]), ('B', [4, 5, 6])]))
Out[57]:
   A  B
0  1  4
1  2  5
2  3  6
```

If you pass `orient='index'`, the keys will be the row labels. In this case, you can also pass the desired column names:

```
In [58]: pd.DataFrame.from_dict(dict([('A', [1, 2, 3]), ('B', [4, 5, 6])]),
   ....:                         orient='index', columns=['one', 'two', 'three'])
   ....:
Out[58]:
   one  two  three
A    1    2      3
B    4    5      6
```

DataFrame.from_records

`DataFrame.from_records` takes a list of tuples or an ndarray with structured dtype. It works analogously to the normal `DataFrame` constructor, except that the resulting DataFrame index may be a specific field of the structured dtype. For example:

```
In [59]: data
Out[59]:
array([(1, 2., b'Hello'), (2, 3., b'World')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])

In [60]: pd.DataFrame.from_records(data, index='C')
Out[60]:
          A    B
C
b'Hello'  1  2.0
b'World'  2  3.0
```

Column selection, addition, deletion

You can treat a DataFrame semantically like a dict of like-indexed Series objects. Getting, setting, and deleting columns works with the same syntax as the analogous dict operations:

```
In [61]: df['one']
Out[61]:
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
```

(continues on next page)

(continued from previous page)

```
In [62]: df['three'] = df['one'] * df['two']

In [63]: df['flag'] = df['one'] > 2

In [64]: df
Out[64]:
   one  two  three   flag
a  1.0  1.0    1.0  False
b  2.0  2.0    4.0  False
c  3.0  3.0    9.0   True
d  NaN  4.0    NaN  False
```

Columns can be deleted or popped like with a dict:

```
In [65]: del df['two']

In [66]: three = df.pop('three')

In [67]: df
Out[67]:
   one   flag
a  1.0  False
b  2.0  False
c  3.0   True
d  NaN  False
```

When inserting a scalar value, it will naturally be propagated to fill the column:

```
In [68]: df['foo'] = 'bar'

In [69]: df
Out[69]:
   one   flag  foo
a  1.0  False  bar
b  2.0  False  bar
c  3.0   True  bar
d  NaN  False  bar
```

When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame's index:

```
In [70]: df['one_trunc'] = df['one'][:2]

In [71]: df
Out[71]:
   one   flag  foo  one_trunc
a  1.0  False  bar        1.0
b  2.0  False  bar        2.0
c  3.0   True  bar        NaN
d  NaN  False  bar        NaN
```

You can insert raw ndarrays but their length must match the length of the DataFrame's index.

By default, columns get inserted at the end. The `insert` function is available to insert at a particular location in the columns:

```
In [72]: df.insert(1, 'bar', df['one'])

In [73]: df
Out[73]:
   one  bar   flag  foo  one_trunc
a  1.0  1.0  False  bar        1.0
b  2.0  2.0  False  bar        2.0
c  3.0  3.0   True  bar        NaN
d  NaN  NaN  False  bar        NaN
```

Assigning new columns in method chains

Inspired by dplyr's `mutate` verb, DataFrame has an *`assign()`* method that allows you to easily create new columns that are potentially derived from existing columns.

```
In [74]: iris = pd.read_csv('data/iris.data')

In [75]: iris.head()
Out[75]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
2          4.7         3.2          1.3         0.2  Iris-setosa
3          4.6         3.1          1.5         0.2  Iris-setosa
4          5.0         3.6          1.4         0.2  Iris-setosa

In [76]: (iris.assign(sepal_ratio=iris['SepalWidth'] / iris['SepalLength'])
   ....:       .head())
   ....:
Out[76]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
```

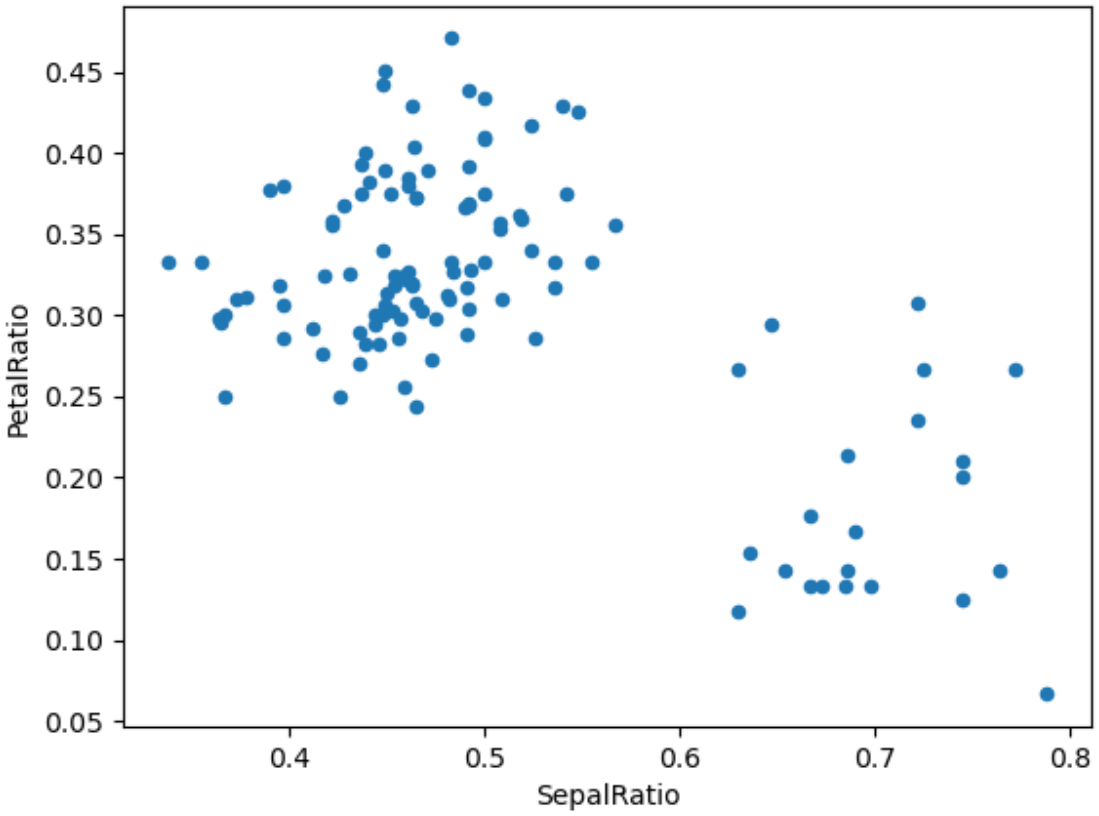
In the example above, we inserted a precomputed value. We can also pass in a function of one argument to be evaluated on the DataFrame being assigned to.

```
In [77]: iris.assign(sepal_ratio=lambda x: (x['SepalWidth'] / x['SepalLength'])).
↪head()
Out[77]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
```

`assign` **always** returns a copy of the data, leaving the original DataFrame untouched.

Passing a callable, as opposed to an actual value to be inserted, is useful when you don't have a reference to the DataFrame at hand. This is common when using `assign` in a chain of operations. For example, we can limit the DataFrame to just those observations with a Sepal Length greater than 5, calculate the ratio, and plot:

```
In [78]: (iris.query('SepalLength > 5')
   ....:       .assign(SepalRatio=lambda x: x.SepalWidth / x.SepalLength,
   ....:               PetalRatio=lambda x: x.PetalWidth / x.PetalLength)
   ....:       .plot(kind='scatter', x='SepalRatio', y='PetalRatio'))
   ....:
Out[78]: <matplotlib.axes._subplots.AxesSubplot at 0x7f5369dff450>
```



Since a function is passed in, the function is computed on the DataFrame being assigned to. Importantly, this is the DataFrame that's been filtered to those rows with sepal length greater than 5. The filtering happens first, and then the ratio calculations. This is an example where we didn't have a reference to the *filtered* DataFrame available.

The function signature for `assign` is simply `**kwargs`. The keys are the column names for the new fields, and the values are either a value to be inserted (for example, a `Series` or NumPy array), or a function of one argument to be called on the `DataFrame`. A *copy* of the original DataFrame is returned, with the new values inserted.

Changed in version 0.23.0.

Starting with Python 3.6 the order of `**kwargs` is preserved. This allows for *dependent* assignment, where an expression later in `**kwargs` can refer to a column created earlier in the same *`assign()`*.

```
In [79]: dfa = pd.DataFrame({"A": [1, 2, 3],
   ....:                     "B": [4, 5, 6]})
   ....:

In [80]: dfa.assign(C=lambda x: x['A'] + x['B'],
```

(continues on next page)