03 - Python_for_Data_Science_Numpy_Pandas_Course_Notes

November 25, 2021

1 Python for Data Science

By: Himalaya Kakshapati

Welcome to Python for Data Science course!

2 NumPy

NumPy is a powerful Python library, which is a short form of 'Numerial Python'. It provides an efficient and easy way to store and operate on dense data. Although NumPy arrays look very much like Python lists, they provide much more efficient storage and data operations as the arrays grow larger in size. NumPy arrays are the core of almost all data science tools in Python.

Once you have installed NumPy on your system, you need to import the library as follows.

[ ]:
```
import numpy
numpy.__version__
```

[ ]:
```
'1.19.5'
```

It is advisable to import `numpy` as an alias as follows.

[ ]:
```
import numpy as np
np.__version__
```

[ ]:
```
'1.19.5'
```

2.1 NumPy Arrays

NumPy arrays provide efficient storage as well as efficient operations on data. As opposed to Python lists (which can store items of different data types), `numpy` arrays can store only one type of items, meaning that all the items in the array need to be of the same type.

2.2 Creating arrays from Python lists

We can create `numpy` array from Python list as follows.

[ ]:
```
my_arr = np.array([2, 4, 8, 16])
print(my_arr)
print(type(my_arr)) # data type is ndarray
```

```
[ 2  4  8 16]
<class 'numpy.ndarray'>
```

If you try to put different data types in the numpy array, numpy tries to upcast if it is possible. Examples follow.

[ ]:
```
np.array([1, 2.4, 3, 5]) # int types upcasted to float
```

[ ]:
```
array([1. , 2.4, 3. , 5. ])
```

[ ]:
```
np.array([3, 2.4, 3, 'mystr']) # int and float upcasted to str
```

[ ]:
```
array(['3', '2.4', '3', 'mystr'], dtype='<U32')
```

We can also explicitly set the data type of the array using the dtype keyword, as follows.

[ ]:
```
np.array([1.25, 2, 3, 4], dtype='int16')
```

[ ]:
```
array([1, 2, 3, 4], dtype=int16)
```

[ ]:
```
np.array([1.25, 2, 3, 4], dtype='float32')
```

[ ]:
```
array([1.25, 2.  , 3.  , 4.  ], dtype=float32)
```

numpy array can be multi-dimensional. Let's create a 2-dimensional numpy array.

[ ]:
```
import numpy as np
np.array([        # a 3 by 3 array
          [1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]
        ])
```

[ ]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

2.3 Creating arrays with built-in methods

Here are some ways we can create arrays with built-in methods.

[ ]:
```
# create an array of zeros with length 5
np.zeros(5, dtype=int)
```

[ ]:
```
array([0, 0, 0, 0, 0])
```

[ ]:
```
# create a 3-dimensional array filled with ones
np.ones((3,5,2), dtype=float)
```

[ ]:
```
array([[[1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.]]])
```

[ ]:
```
# create an array filled with a linear sequence which starts at 1 and ends at 30
# with a step of 3
np.arange(1, 31, 3)
```

[ ]:
```
array([ 1,  4,  7, 10, 13, 16, 19, 22, 25, 28])
```

[ ]:
```
# create an array of 5 values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
```

[ ]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

[ ]:
```
# create a 3x3 array of uniformly distributed random values between 0 and 1
np.random.random((3,3))
```

[ ]:
```
array([[0.42067217, 0.70291168, 0.1755839 ],
       [0.30032511, 0.40328841, 0.48248901],
       [0.44021045, 0.74621534, 0.30363889]])
```

[ ]:
```
# create a 3x3 array of normally distributed random values with mean (average)
# 0 and standard deviation 1
np.random.normal(0, 1, (3, 3))
```

[ ]:
```
array([[-1.23726613, -0.56923989,  0.9014251 ],
       [-1.91113922, -0.25447295, -0.20999147],
       [-0.14897987, -0.81925242,  0.29710332]])
```

[ ]:
```
# create a 3x3 array of random integers between 0 and 10 (including 0 and␣
↪excluding 10),
# in notation [0, 10)
```

```
np.random.randint(0, 10, (3,3))
```

[ ]:
```
array([[3, 9, 9],
       [8, 3, 3],
       [4, 2, 7]])
```

[ ]:
```
# create a 3x3 identity matrix
np.eye(3)
```

[ ]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

[ ]:
```
# create an uninitialized array of 3 integers. The values will be whatever␣
↪happens
# to exist already in that memory location
np.empty(10)
```

[ ]:
```
array([4.65914834e-310, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
       0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
       0.00000000e+000, 0.00000000e+000])
```

2.4 NumPy standard data types

As NumPy is coded in the C programming language, the standard data types in `numpy` are similar to C data types.

The source of the following table is: Python Data Science Handbook (By: Jake VanderPlas)

2.5 Basics of NumPy Arrays

Let's look at some examples of how we can manipulate `numpy` arrays in order to access data and subarrays, split, reshape, join arrays etc.

2.5.1 NumPy Array Attributes

Let's first define 3 random arrays:

- One-dimensional array
- Two-dimensional array
- Three-dimensional array

[ ]:
```
import numpy as np
np.random.seed(0) # seed ensures that we get the same random numbers each time␣
↪(reproducibility)

# one-dim array
```

```
x1 = np.random.randint(10, size=6)
x2 = np.random.randint(10, size=(4,3))
x3 = np.random.randint(10, size=(2,3,4))
x1
```

[ ]: array([5, 0, 3, 3, 7, 9])

The above arrays have the following attributes.

- `ndim` => Number of dimensions
- `shape` => Size of each dimension
- `size` => Total size of the array

[ ]:
```
# number of dimensions
print(x1.ndim)
print(x2.ndim)
print(x3.ndim)
```

```
1
2
3
```

[ ]:
```
# shape
print(x1.shape)
print(x2.shape)
print(x3.shape)
```

```
(6,)
(4, 3)
(2, 3, 4)
```

[ ]:
```
# size
print(x1.size)
print(x2.size)
print(x3.size)
```

```
6
12
24
```

Another useful attribute of `numpy` array is `dtype`, which is the data type of the array.

[ ]:
```
x1.dtype
```

[ ]: dtype('int64')

Other attributes of `numpy` array:

- `itemsize` => size in bytes of each item (element) in the array
- `nbytes` => total size of the array in bytes

Evidently, `nbytes` = `itemsize` * `size`

```
[ ]: print("x2.itemsize =", x1.itemsize)
     print("x2.nbytes =", x1.nbytes)
```

```
x2.itemsize = 8
x2.nbytes = 48
```

```
[ ]: x2.nbytes == x2.itemsize * x2.size
```

```
[ ]: True
```

2.5.2 Array Indexing: Accessing Single Elements

For one-dimensional arrays, array indexing is the same as list indexing.

```
[ ]: print(x1)
     print(x1[0])
```

```
[5 0 3 3 7 9]
5
```

You can use negative indexing as well, just as in lists.

For multi-dimensional arrays, you can access items with comma-separated tuple of indices. Example:

```
[ ]: print(x2)
     print("item at row 1, column 1: ", x2[0,0])
     print("item at row 3, column 3: ", x2[2,-1])
```

```
[[3 5 2]
 [4 7 6]
 [8 8 1]
 [6 7 7]]
item at row 1, column 1:  3
item at row 3, column 3:  1
```

You can update a value in the array using indexing as shown below.

```
[ ]: x2[2,-1] = 3.14159
     print(x2)
     print("updated item at row 3, column 3: ", x2[2,-1]) # 3.14159 will be trucated␣
      ↪to 3
```

```
[[3 5 2]
 [4 7 6]
 [8 8 3]
 [6 7 7]]
updated item at row 3, column 3:  3
```

2.5.3 Array Slicing: Accessing Subarrays

We can use square brackets to access subarrays with slice notation (indices/step separated by colons).

The syntax is: x[start index : stop index + 1 : step] If no index and step are given (e.g. x[::]), then by default,

- start index = 0
- stop index = last index
- step = 1

[ ]:
```
# one-dim array
import numpy as np
x = np.arange(10)
x
```

[ ]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[ ]:
```
# slicing
print("x[:7] => ", x[:7]) # access index 0 through 6
print("x[7:] => ", x[7:]) # access index 7 through to end index
print("x[3:8] => ", x[3:8]) # access index 3 through 7
print("x[::2] => ", x[::2]) # access every other element starting from the first
print("x[2::2] => ", x[2::2]) # access every other element starting from the␣
↪index 2
print("x[::] => ", x[::]) # access all elements

# if the step value is negative, elements are accessed from start_index to␣
↪stop_index
# where start_index > stop_index, and step is decremented by the step size
print("x[7:3:-1] => ", x[7:3:-1]) # index 7 to index 4 decremented by 1
```

```
x[:7] =>  [0 1 2 3 4 5 6]
x[7:] =>  [7 8 9]
x[3:8] =>  [3 4 5 6 7]
x[::2] =>  [0 2 4 6 8]
x[2::2] =>  [2 4 6 8]
x[::] =>  [0 1 2 3 4 5 6 7 8 9]
x[7:3:-1] =>  [7 6 5 4]
```

Accessing Multi-dimensional subarrays

[ ]:
```
x2
```

[ ]:
```
array([[3, 5, 2],
       [4, 7, 6],
       [8, 8, 3],
       [6, 7, 7]])
```

[ ]:
```
x2[:3, :2] # rows => index 0 to 2; columns => index 0 to 1
```

[ ]:
```
array([[3, 5],
       [4, 7],
       [8, 8]])
```

[ ]:
```
x2[::2,:] # every other row; all columns
```

[ ]:
```
array([[3, 5, 2],
       [8, 8, 3]])
```

[ ]:
```
# reverse altogether - both rows and columns
x2[::-1, ::-1]
```

[ ]:
```
array([[7, 7, 6],
       [3, 8, 8],
       [6, 7, 4],
       [2, 5, 3]])
```

Accessing array rows and columns

[ ]:
```
x2
```

[ ]:
```
array([[3, 5, 2],
       [4, 7, 6],
       [8, 8, 3],
       [6, 7, 7]])
```

[ ]:
```
x2[:,1] # second column
```

[ ]:
```
array([5, 7, 8, 7])
```

[ ]:
```
x2[1,:] # second row
```

[ ]:
```
array([4, 7, 6])
```

2.5.4 Reshaping Arrays

You can use the reshape() method to reshape arrays.

[ ]:
```
# reshape a one-dim array to 3x3 matrix
np.arange(1,10).reshape((3,3)) # size of original array must match the size of
↪the reshaped array
```

[ ]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

You can reshape a one-dimensional array into a two-dimensional row or column matrix.

[ ]:
```
import numpy as np
x = np.arange(5)
x # one-dim array
```

[ ]:
```
array([0, 1, 2, 3, 4])
```

[ ]:
```
x.shape
```

[ ]:
```
(5,)
```

[ ]:
```
# row vector using reshape
x.reshape((1,5)) # reshape to 1x5 matrix (row matrix) -- 2-dim array
```

[ ]:
```
array([[0, 1, 2, 3, 4]])
```

[ ]:
```
# row vector using newaxis
x[np.newaxis,:]
```

[ ]:
```
array([[0, 1, 2, 3, 4]])
```

[ ]:
```
# column vector using reshape
x.reshape((5,1)) # reshape to 5x1 matrix (column matrix)
```

[ ]:
```
array([[0],
       [1],
       [2],
       [3],
       [4]])
```

[ ]:
```
# column vector using newaxis
x[:, np.newaxis]
```

[ ]:
```
array([[0],
       [1],
       [2],
       [3],
       [4]])
```

2.5.5 Array Concatenation and Splitting

We can combine multiple arrays into one, or split a single array to multiple arrays.

Concatenation (joining) of arrays

We can concatenate arrays can be done with the following routines.

- np.concatenate

- np.vstack
- np.hstack

[ ]:
```
# np.concatenate
import numpy as np
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
np.concatenate([x, y])
```

[ ]:
```
array([1, 2, 3, 4, 5, 6])
```

[ ]:
```
# concatenate more than 2 arrays using np.concatenate
z = [7, 8, 9]
np.concatenate([x, y, z])
```

[ ]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[ ]:
```
# concatenate 2-dim arrays along first axis (rows)
arr2d = np.array([
                  [1, 2, 3],
                  [4, 5, 6]
                ])
np.concatenate([arr2d, arr2d]) # axis=0 by default => means concatente along␣
 ↪axis 0 (row)
```

[ ]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

[ ]:
```
# concatenate 2-dim arrays along second axis (axis 1 => column)
np.concatenate([arr2d, arr2d], axis = 1)
```

[ ]:
```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

Concatenate using np.vstack (vertical stack) and np.hstack (horizontal stack)

[ ]:
```
# np.vstack
np.vstack([x, arr2d])
```

[ ]:
```
array([[1, 2, 3],
       [1, 2, 3],
       [4, 5, 6]])
```

[ ]:
```
# np.hstack
y = [[99],
     [99]]
```

```
np.hstack([y, arr2d])
```

[ ]:
```
array([[99,  1,  2,  3],
       [99,  4,  5,  6]])
```

Splitting of Arrays

Splitting is the opposite of concatenation. The following functions can be used for splitting the array.

- np.split
- np.hsplit
- np.vsplit

[ ]:
```
# np.split
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5]) # np.split(array, list of indices split␣
 ↪points)
print(x1, x2, x3)
```

```
[1 2 3] [99 99] [3 2 1]
```

[ ]:
```
grid = np.arange(16).reshape((4, 4))
grid
```

[ ]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

[ ]:
```
# np.vsplit
upper, lower = np.vsplit(grid, [2]) # np.vsplit(arr, split index)
print(upper)
print(lower)
```

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

[ ]:
```
# np.hsplit
left, right = np.hsplit(grid, [2]) # np.hsplit(arr, split index)
print(left)
print(right)
```

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
```

```
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

2.6 Computation on NumPy Arrays: Universal Functions

The reason NumPy is so important in the world of Python Data Science is that it provides an easy and flexible interface to optimized computation with arrays of data. What make them so fast are the vectorized operations, which are generally implemented through NumPy's *Universal Functions* (ufuncs).

[ ]:
```
# computing reciprocals of an array using a loop
import numpy as np
np.random.seed(0)

def compute_reciprocals(values):
  output = np.empty(len(values))
  for i in range(len(values)):
    output[i] = 1.0 / values[i]
  return output

values = np.random.randint(1, 10, size=5)
compute_reciprocals(values)
```

[ ]:
```
array([0.16666667, 1.        , 0.25      , 0.25      , 0.125     ])
```

[ ]:
```
big_array = np.random.randint(1, 100, size = 1000000) # array with 1 million␣
 ↪elements
%timeit compute_reciprocals(big_array)
```

```
1 loop, best of 5: 2.37 s per loop
```

As you can see, the implementation of computing reciprocals by using loops is extremely slow (few seconds per loop).

Introducing UFuncs

Now, let's implement the same operation as above (computing reciprocals of each element of an array) using *vectorized* operation.

[ ]:
```
%timeit reciprocals = 1.0/big_array # vectorized
```

```
100 loops, best of 5: 2.64 ms per loop
```

As you can see, the vectorized approach is orders of magnitude faster than the loop version (few milliseconds vs. few seconds).

The vectorized operations in NumPy are implemented using *ufuncs*.

UFuncs can be

- Unary
- Binary

Below, you can see some examples of UFuncs.

[ ]:
```
x = np.arange(4)

# vectorized operations using ufuncs
print("x =      ", x)
print("x + 5 =  ", x + 5)
print("x - 5 =  ", x - 5)
print("x * 5 =  ", x * 5)
print("x / 5 =  ", x / 5)
print("x // 5 = ", x // 5) # floor division
```

```
x =       [0 1 2 3]
x + 5 =   [5 6 7 8]
x - 5 =   [-5 -4 -3 -2]
x * 5 =   [ 0  5 10 15]
x / 5 =   [0.  0.2 0.4 0.6]
x // 5 =  [0 0 0 0]
```

[ ]:
```
print(" -x =     ", -x) # negation (unary operation)
print(" x ** 2 = ", x ** 2) # exponentiation
print(" x % 2 =  ", x % 2) # modulus
```

```
 -x =      [ 0 -1 -2 -3]
 x ** 2 =  [0 1 4 9]
 x % 2 =   [0 1 0 1]
```

[ ]:
```
# standard order of operation is respected
-(0.5*x + 5) ** 3
```

[ ]:
```
array([-125.   , -166.375, -216.   , -274.625])
```

(source: Python Data Science Handbook, Jake Vanderplas)

Just as NumPy understands Python's built-in arithmetic operators, it also understands Python's built-in absolute value function.

[ ]:
```
x = np.array([-2, 3, 10, -3, -1])
abs(x) # built-in abs function
```

[ ]:
```
array([ 2,  3, 10,  3,  1])
```

[ ]:
```
print( np.absolute(x) ) # numpy absolute function
print( np.abs(x) ) # alias of np.absolute
```

```
[ 2  3 10  3  1]
[ 2  3 10  3  1]
```

[ ]:
```
# ufuncs can also deal with data involving complex numbers
# absolute value of complex data returns the magnitude
x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
np.abs(x)
```

[ ]:
```
array([5., 5., 2., 1.])
```

[ ]:
```
abs(x) # built-in abs function on complex data
```

[ ]:
```
array([5., 5., 2., 1.])
```

Trigonometric Functions

[ ]:
```
theta = np.linspace(0, np.pi, 3) # in degrees => 0, 90, 180
theta
```

[ ]:
```
array([0.        , 1.57079633, 3.14159265])
```

[ ]:
```
print("sin(theta) = ", np.sin(theta)) # ans: [0, 1, 0]
print("cos(theta) = ", np.cos(theta)) # ans: [1, 0, -1]
print("tan(theta) = ", np.tan(theta)) # ans: [0, infinity, 0]
```

```
sin(theta) =  [0.0000000e+00 1.0000000e+00 1.2246468e-16]
cos(theta) =  [ 1.000000e+00  6.123234e-17 -1.000000e+00]
tan(theta) =  [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

Exponents and logarithms

[ ]:
```
x = [0, 1, 2, 3]
print("x = ", x)
print("e^x = ", np.exp(x))
print("2^x = ", np.exp2(x))
print("3^x = ", np.power(3,x))
```

```
x =  [0, 1, 2, 3]
e^x =  [ 1.          2.71828183  7.3890561  20.08553692]
2^x =  [1. 2. 4. 8.]
3^x =  [ 1  3  9 27]
```

[ ]:
```
# logarithms
x = [1, 2, 4, 10]
print("x = ", x)
print("ln(x) = ", np.log(x)) # np.log gives the natural log (wrt e)
print("log2(x) = ", np.log2(x)) # log base 2
print("log10(x) = ", np.log10(x)) # log base 10
```

```
x =  [1, 2, 4, 10]
ln(x) =  [0.         0.69314718 1.38629436 2.30258509]
log2(x) =  [0.         1.         2.         3.32192809]
log10(x) =  [0.         0.30103    0.60205999 1.        ]
```

2.6.1 Advanced UFunc features

Aggregates

For binary ufuncs, there are some aggregates that can be computed directly from the object. Examples:

- reduce
- accumulate

`reduce` repeatedly applies a given operation to the elements of an array until only a single result remains.

[ ]:
```python
import numpy as np
arr = np.arange(1,10)
np.add.reduce(arr) # sum of all integers from 1 to 9
```

[ ]:
```
45
```

[ ]:
```python
np.multiply.reduce(arr) # product of all integers from 1 to 9
```

[ ]:
```
362880
```

If we want to store the intermediate results of all the computations, we use `accumulate`.

[ ]:
```python
np.add.accumulate(arr)
```

[ ]:
```
array([ 1,  3,  6, 10, 15, 21, 28, 36, 45])
```

Outer Products

Any ufunc can create the output of all pairs of two diffent inputs using the `outer` method.

We can create a multiplication table using `outer` as follows.

[ ]:
```python
x = np.arange(2, 11)
np.multiply.outer(x,x) # outer product of (u, v) => u*transpose(v); assuming u␣
↪and v are column vectors
```

[ ]:
```
array([[  4,   6,   8,  10,  12,  14,  16,  18,  20],
       [  6,   9,  12,  15,  18,  21,  24,  27,  30],
       [  8,  12,  16,  20,  24,  28,  32,  36,  40],
       [ 10,  15,  20,  25,  30,  35,  40,  45,  50],
       [ 12,  18,  24,  30,  36,  42,  48,  54,  60],
       [ 14,  21,  28,  35,  42,  49,  56,  63,  70],
       [ 16,  24,  32,  40,  48,  56,  64,  72,  80],
```

```
[ 18,  27,  36,  45,  54,  63,  72,  81,  90],
[ 20,  30,  40,  50,  60,  70,  80,  90, 100]])
```

2.6.2 Aggregations: Min, Max, and Everything in-between

Summing the Values in an Array

[ ]:
```
# sum of all elements in an array using built-in Python sum function
import numpy as np
arr = np.random.random(50)
sum(arr)
```

[ ]:
```
22.935074583519384
```

[ ]:
```
# sum of all elements in an array using numpy sum function
np.sum(arr)
```

[ ]:
```
22.935074583519388
```

Let's see which one is faster.

[ ]:
```
big_arr = np.random.rand(1000000)
%timeit sum(big_arr)
%timeit np.sum(big_arr)
```

```
10 loops, best of 5: 171 ms per loop
1000 loops, best of 5: 414 µs per loop
```

As you can see, the numpy sum is orders of magnitude faster (microseconds vs. milliseconds).

Minimum and Maximum

Python has built-in `min` and `max` functions.

[ ]:
```
min(big_arr), max(big_arr)
```

[ ]:
```
(7.071203171893359e-07, 0.9999997207656334)
```

Numpy has `min` and `max` functions as well, which are much faster than their built-in counterparts.

[ ]:
```
np.min(big_arr), np.max(big_arr)
```

[ ]:
```
(7.071203171893359e-07, 0.9999997207656334)
```

[ ]:
```
%timeit min(big_arr)
%timeit np.min(big_arr) # much faster
```

```
10 loops, best of 5: 105 ms per loop
The slowest run took 4.17 times longer than the fastest. This could mean that an
```

```
intermediate result is being cached.
1000 loops, best of 5: 332 µs per loop
```

Multidimensional Aggregates

A common type of aggregation operation is an aggregate along a row or column.

[ ]:
```python
import numpy as np
m_arr = np.random.random((3,4))
m_arr
```

[ ]:
```
array([[0.37438374, 0.13285138, 0.53221783, 0.94833657],
       [0.16341688, 0.74628308, 0.53739875, 0.67408516],
       [0.88503042, 0.25803571, 0.03215288, 0.01444397]])
```

[ ]:
```python
# sum of all the elements
print(m_arr.sum())
print(np.sum(m_arr))
```

```
5.298636376897056
5.298636376897056
```

[ ]:
```python
# min of rows
m_arr.min(axis=1) # axis => the axis that is going to be collapsed
```

[ ]:
```
array([0.13285138, 0.16341688, 0.01444397])
```

[ ]:
```python
# max of columns
m_arr.max(axis=0)
```

[ ]:
```
array([0.78670556, 0.71400043, 0.84906741, 0.96649509])
```

[ ]:
```python
# alternatively
np.max(m_arr, axis=0)
```

[ ]:
```
array([0.78670556, 0.71400043, 0.84906741, 0.96649509])
```

(source: Python Data Science Handbook, By Jake Vanderplas)

The NaN-safe versions compute the results ignoring missing values.

2.7 Example: Average Height of US Presidents

Let's do some data analysis on the data contained in the file `president_heights.csv`.

[ ]:
```
! head -5 /content/drive/MyDrive/Python\ Training/president_heights.csv # head␣
↪gives the top rows; -5 means that top 5 rows will be shown
```

```
order,name,height(cm)
1,George Washington,189
2,John Adams,170
3,Thomas Jefferson,189
4,James Madison,163
```

Let us use the Pandas package (covered later) to read the file and extract information.

[ ]:
```
import numpy as np
import pandas as pd
data = pd.read_csv('/content/drive/MyDrive/Python Training/president_heights.
↪csv')
data[:5] # show first 5 rows
```

[ ]:
```
   order                name  height(cm)
0      1   George Washington         189
1      2          John Adams         170
2      3    Thomas Jefferson         189
3      4       James Madison         163
4      5        James Monroe         183
```

[ ]:
```
heights = np.array(data['height(cm)']) # get the data in height column and make␣
↪a numpy array
heights
```

[ ]:
```
array([189, 170, 189, 163, 183, 171, 185, 168, 173, 183, 173, 173, 175,
       178, 183, 193, 178, 173, 174, 183, 183, 168, 170, 178, 182, 180,
       183, 178, 182, 188, 175, 179, 183, 193, 182, 183, 177, 185, 188,
       188, 182, 185])
```

[ ]:
```
type(heights)
```

[ ]:
```
numpy.ndarray
```

Now that we have a data array, we can do a summary statistics on the data.

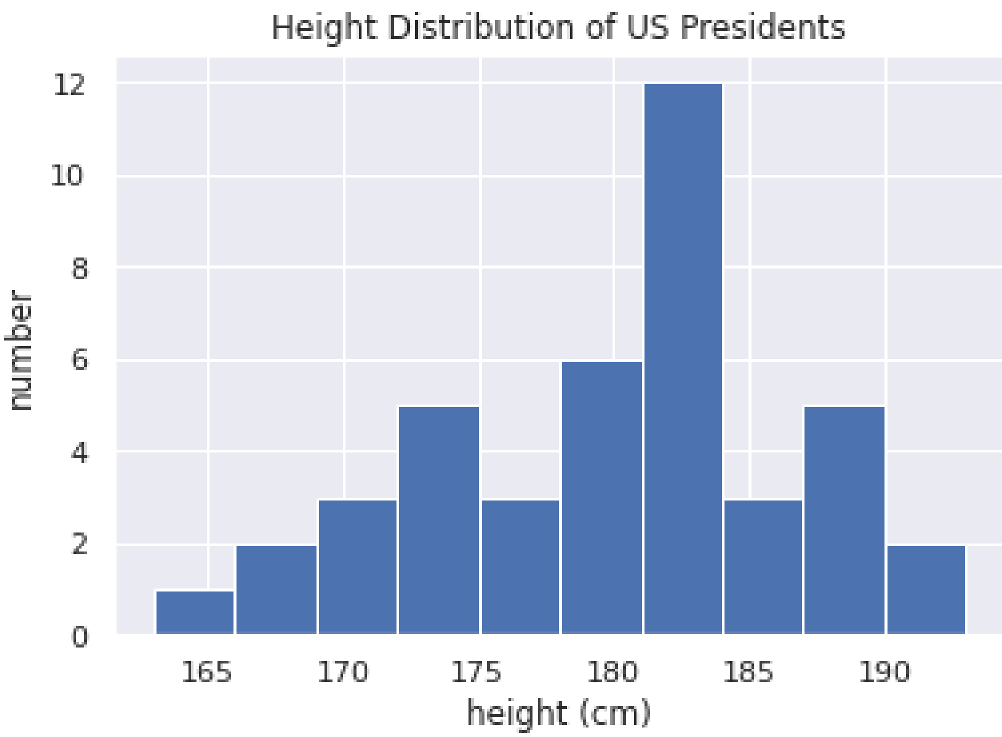
[ ]:
```
print("Mean (average) height:\t",heights.mean())
print("Standard Deviation:\t",heights.std())
print("Minimum height:\t\t",heights.min())
print("Maximum height:\t\t",heights.max())
```

```
Mean (average) height:   179.73809523809524
Standard Deviation:      6.931843442745892
Minimum height:          163
Maximum height:          193
```

Now, let's perform some visualizations on the data using Matplotlib library (covered later).

[ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
```

[ ]:
```
plt.hist(heights) # plot the histogram
plt.title('Height Distribution of US Presidents') # title of the graph
plt.xlabel('height (cm)') # label of the x-axis
plt.ylabel('number'); # label of the y-axis
```



2.8 Computation on Arrays: Broadcasting

Previously, you saw how NumPy's universal functions (ufuncs) can be used to vectorize operations, thereby eliminating the need for slow loops.

Another means of vectorizing the operations is by using NumPy's *broadcasting* functionality.

Simply put, *broadcasting* is a set of rules for applying binary ufuncs (addition, subtraction, multiplication etc.) on arrays of different sizes.

[ ]:
```
import numpy as np

x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
```

```
x + y # ufunc
```

[ ]: array([5, 7, 9])

What if you want to perform such operations on two arrays of different sizes?

Broadcasting allows these types of operations to be performed on arrays of different sizes.

[ ]:
```
# add a scalar (think of zero dimensional array) to an array
x + 3 # array + scalar value => Think of it as: [1, 2, 3] + [3, 3, 3]
```

[ ]: array([4, 5, 6])

Now, let's look at broadcasting in higher dimensions.

[ ]:
```
md_arr = np.ones((3,3))
md_arr # 3x3 array
```

[ ]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

[ ]:
```
x # 1 dimensional array
```

[ ]: array([1, 2, 3])

[ ]:
```
md_arr + x # broadcasting
```

[ ]:
```
array([[2., 3., 4.],
       [2., 3., 4.],
       [2., 3., 4.]])
```

More complicated cases can involve broadcasting of both arrays.

[ ]:
```
a = np.arange(3) # one dimensional array
b = np.arange(3)[:, np.newaxis] # 3x1 array

print(a)
print(b)
```

```
[0 1 2]
[[0]
 [1]
 [2]]
```

[ ]:
```
a + b # both arrays broadcasted
```

[ ]:
```
array([[0, 1, 2],
       [1, 2, 3],
```

```
        [2, 3, 4]])
```

2.8.1 Broadcasting in Practice

Centering an array

Consider that you have an array of 10 observations, each of which consists of 3 values.

[ ]:
```
X = np.random.random((10,3)) # 10x3 array
X[:5] # show first 5 rows
```

[ ]:
```
array([[0.8404517 , 0.71815808, 0.63212618],
       [0.85962491, 0.06822834, 0.71826061],
       [0.74547901, 0.35812467, 0.64126629],
       [0.08713306, 0.33573899, 0.07597282],
       [0.81600654, 0.77711271, 0.78033451]])
```

[ ]:
```
X_mean = X.mean(axis=0) # mean of each column (feature)
X_mean
```

[ ]:
```
array([0.39823522, 0.56460162, 0.52674091])
```

We can center the X array by subtracting the mean.

[ ]:
```
# X_mean is broadcasted in the following operation
X_centered = X - X_mean # X is centered, meaning that the mean of each column␣
↪will be zero
```

The mean of each column (feature) should be ideally 0.

[ ]:
```
X_centered.mean(axis=0) # mean of columns close to zero
```

[ ]:
```
array([ 4.44089210e-17,  7.77156117e-17, -7.77156117e-17])
```

2.9 Comparisons, Masks, and Boolean Logic

Let's see how we can use Boolean masks to examine and manipulate values in NumPy arrays.

Comparison Operators as ufuncs

NumPy also implements comparison operators such as `<` and `>` as element-wise ufuncs. The result of these operations is always an array with a Boolean data type.

[ ]:
```
x = np.array([1, 2, 3, 4, 5])
x
```

[ ]:
```
array([1, 2, 3, 4, 5])
```

[ ]:
```
x < 3
```

[ ]:
```
array([ True,  True, False, False, False])
```

[ ]:
```
x > 2
```

[ ]:
```
array([False, False,  True,  True,  True])
```

[ ]:
```
x != 4
```

[ ]:
```
array([ True,  True,  True, False,  True])
```

[ ]:
```
# element-by-element comparison of two arrays, to include compound expressions
(2 * x) == (x ** 2)
```

[ ]:
```
array([False,  True, False, False, False])
```

Just like with arithmetic operators, the comparison operators are also implemented as ufuncs in Numpy.

Example: when you run `x < 2`, internally Numpy uses `np.less(x, 2)`

(source: Python Data Science Handbook, By: Jake Vanderplas)

Just as with arithmetic ufuncs, comparison ufuncs works on array of any shape and size.

[ ]:
```
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
x
```

[ ]:
```
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
```

[ ]:
```
x <= 5
```

[ ]:
```
array([[ True,  True,  True,  True],
       [False, False,  True,  True],
       [ True,  True, False, False]])
```

2.9.1 Working with Boolean Arrays

[ ]:
```
x # 3x4 array
```

[ ]:
```
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
```

```
       [2, 4, 7, 6]])
```

Counting entries

To count the number of `True` entries in a Boolean array, we can use `np.count_nonzero` function.

[ ]:
```
# Counting elements less than 5
np.count_nonzero(x < 5)
```

[ ]:
```
6
```

[ ]:
```
# np.sum also works since False is considered to be 0 and True is considered to
 ↪be 1
np.sum(x < 5)
```

[ ]:
```
6
```

[ ]:
```
# number of elements less than 5 in each row
np.count_nonzero(x < 5, axis=1)
```

[ ]:
```
array([3, 1, 2])
```

[ ]:
```
# are there any values greater than 8?
np.any(x > 8)
```

[ ]:
```
True
```

[ ]:
```
# are there any values less than zero?
np.any(x < 0)
```

[ ]:
```
False
```

[ ]:
```
# are all values less than 10?
np.all(x < 10)
```

[ ]:
```
True
```

[ ]:
```
# are all values in each row less than 8?
np.all(x < 8, axis=1)
```

[ ]:
```
array([ True, False,  True])
```

Boolean Operators

Bit-wise logical operators

- & (and)
- (or)

- ^ (xor)
- ~ (not)

[ ]:
```
x
```

[ ]:
```
array([1, 2, 3, 4, 5])
```

[ ]:
```
# number of elements greater than 2 and less than 5
np.sum((x > 2) & (x < 5)) # make sure the conditions are within parentheses
```

[ ]:
```
2
```

(source: Python for Data Science Handbook, By: Jake Vanderplas)

2.9.2 Boolean Arrays as Masks

We can use Boolean arrays as masks to select particular subsets of the data.

[ ]:
```
import numpy as np

x = np.random.randint(0, 10, (3,4)) # 3x4 array
x
```

[ ]:
```
array([[6, 7, 6, 6],
       [2, 5, 3, 3],
       [4, 6, 4, 2]])
```

[ ]:
```
x > 5
```

[ ]:
```
array([[ True,  True,  True,  True],
       [False, False, False, False],
       [False,  True, False, False]])
```

Now to select the values greater than 5, we can simply index on the Boolean array shown above.

[ ]:
```
x[ x > 5 ]
```

[ ]:
```
array([6, 7, 6, 6, 6])
```

2.9.3 Example: Counting Rainy Days

Let's do some data analysis on daily rainfall statistics for the city of Seattle in 2014.

We will use Pandas and Matplotlib as follows.

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

# extract rainfall in inches as a NumPy array
data = pd.read_csv('/content/drive/MyDrive/Python Training/
↪PythonDataScienceHandbook-master/notebooks/data/Seattle2014.csv')
data.head()
```

[ ]:
```
             STATION                           STATION_NAME  …  WT02
WT03
0  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US  … -9999
-9999
1  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US  … -9999
-9999
2  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US  … -9999
-9999
3  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US  … -9999
-9999
4  GHCND:USW00024233  SEATTLE TACOMA INTERNATIONAL AIRPORT WA US  … -9999
-9999

[5 rows x 17 columns]
```

[ ]:
```
rainfall = data['PRCP'].values
rainfall
```

[ ]:
```
array([  0,  41,  15,   0,   0,   3, 122,  97,  58,  43, 213,  15,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   5,   0,   0,   0,   0,
         0,  89, 216,   0,  23,  20,   0,   0,   0,   0,   0,   0,  51,
         5, 183, 170,  46,  18,  94, 117, 264, 145, 152,  10,  30,  28,
        25,  61, 130,   3,   0,   0,   0,   5, 191, 107, 165, 467,  30,
         0, 323,  43, 188,   0,   0,   5,  69,  81, 277,   3,   0,   5,
         0,   0,   0,   0,   0,  41,  36,   3, 221, 140,   0,   0,   0,
         0,  25,   0,  46,   0,   0,  46,   0,   0,   0,   0,   0,   0,
         5, 109, 185,   0, 137,   0,  51, 142,  89, 124,   0,  33,  69,
         0,   0,   0,   0,   0, 333, 160,  51,   0,   0, 137,  20,   5,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  38,
         0,  56,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,  18,  64,   0,   5,  36,  13,   0,
         8,   3,   0,   0,   0,   0,   0,   0,  18,  23,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   3, 193,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   5,   0,   0,   0,   0,   0,   0,   0,
         0,   5, 127, 216,   0,  10,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,  84,  13,   0,  30,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   5,
         3,   0,   0,   0,   3, 183, 203,  43,  89,   0,   0,   8,   0,
```

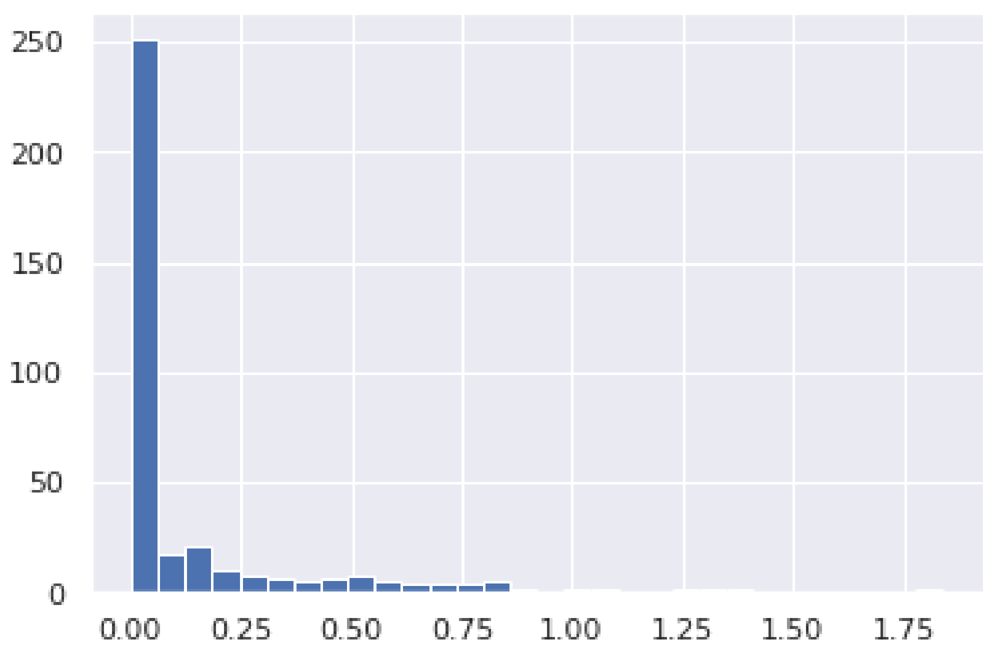
```
 0,   0,   0,   0,   0,   0,   0,   0,   0,   3,  74,   0,  76,
71,  86,   0,  33, 150,   0, 117,  10, 320,  94,  41,  61,  15,
 8, 127,   5, 254, 170,   0,  18, 109,  41,  48,  41,   0,   0,
51,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  36, 152,
 5, 119,  13, 183,   3,  33, 343,  36,   0,   0,   0,   0,   8,
30,  74,   0,  91,  99, 130,  69,   0,   0,   0,   0,   0,  28,
130,  30, 196,   0,   0, 206,  53,   0,   0,  33,  41,   0,   0,
 0])
```

[ ]:
```
inches = rainfall / 254 # converting 1/mm => inches (1 inch = 25.4 mm)
inches[:10]
```

[ ]:
```
array([0.        , 0.16141732, 0.05905512, 0.        , 0.        ,
       0.01181102, 0.48031496, 0.38188976, 0.22834646, 0.16929134])
```

Let's create a histogram plot of the rainfall data.

[ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # set plot styles

plt.hist(inches, bins=30); # bins defines the number of equal-width bins in the␣
 ↪range.
```



The above histogram shows that there was no rain (tallest near 0) for most of the days in 2014.

Now, let's try to find out how many days had a rainfall of between 0.5 and 1.0 inches in Seattle in

2014.

[ ]:
```
np.sum( (inches > 0.5) & (inches < 1.0) ) # bitwise 'and' operator used
```

[ ]:
```
29
```

We see that there were 29 days which had a rainfall between 0.5 and 1.0 inches.

Let's print some other queries on the data.

[ ]:
```
print("Number of days without rain:             ", np.sum(inches == 0))
print("Number of days with rain:                ", np.sum(inches != 0))
print("Number of days with more than 0.5 inches:", np.sum(inches > 0.5))
```

```
Number of days without rain:              215
Number of days with rain:                 150
Number of days with more than 0.5 inches: 37
```

Use of Boolean Masks

[ ]:
```
# mask of all rainy days
rainy = (inches > 0)
rainy[:20]
```

[ ]:
```
array([False,  True,  True, False, False,  True,  True,  True,  True,
        True,  True,  True, False, False, False, False, False, False,
       False, False])
```

[ ]:
```
# construct a mask of all summer days (June 21st is the 172nd day)
summer = (np.arange(365) - 172 < 90) & (np.arange(365) - 172 > 0)
```

[ ]:
```
summer
```

[ ]:
```
array([False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
```

```
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True,  True,  True,  True,  True,  True,  True,
         True, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False])
```

[ ]:
```
inches[rainy]
```

[ ]:
```
array([0.16141732, 0.05905512, 0.01181102, 0.48031496, 0.38188976,
       0.22834646, 0.16929134, 0.83858268, 0.05905512, 0.01968504,
       0.3503937 , 0.8503937 , 0.09055118, 0.07874016, 0.2007874 ,
       0.01968504, 0.72047244, 0.66929134, 0.18110236, 0.07086614,
       0.37007874, 0.46062992, 1.03937008, 0.57086614, 0.5984252 ,
       0.03937008, 0.11811024, 0.11023622, 0.0984252 , 0.24015748,
       0.51181102, 0.01181102, 0.01968504, 0.7519685 , 0.42125984,
       0.6496063 , 1.83858268, 0.11811024, 1.27165354, 0.16929134,
       0.74015748, 0.01968504, 0.27165354, 0.31889764, 1.09055118,
       0.01181102, 0.01968504, 0.16141732, 0.14173228, 0.01181102,
       0.87007874, 0.5511811 , 0.0984252 , 0.18110236, 0.18110236,
       0.01968504, 0.42913386, 0.72834646, 0.53937008, 0.2007874 ,
       0.55905512, 0.3503937 , 0.48818898, 0.12992126, 0.27165354,
       1.31102362, 0.62992126, 0.2007874 , 0.53937008, 0.07874016,
       0.01968504, 0.1496063 , 0.22047244, 0.07086614, 0.2519685 ,
       0.01968504, 0.14173228, 0.0511811 , 0.03149606, 0.01181102,
       0.07086614, 0.09055118, 0.01181102, 0.75984252, 0.01968504,
       0.01968504, 0.5       , 0.8503937 , 0.03937008, 0.33070866,
       0.0511811 , 0.11811024, 0.01968504, 0.01181102, 0.01181102,
```

```
       0.72047244, 0.7992126 , 0.16929134, 0.3503937 , 0.03149606,
       0.01181102, 0.29133858, 0.2992126 , 0.27952756, 0.33858268,
       0.12992126, 0.59055118, 0.46062992, 0.03937008, 1.25984252,
       0.37007874, 0.16141732, 0.24015748, 0.05905512, 0.03149606,
       0.5       , 0.01968504, 1.        , 0.66929134, 0.07086614,
       0.42913386, 0.16141732, 0.18897638, 0.16141732, 0.2007874 ,
       0.14173228, 0.5984252 , 0.01968504, 0.46850394, 0.0511811 ,
       0.72047244, 0.01181102, 0.12992126, 1.3503937 , 0.14173228,
       0.03149606, 0.11811024, 0.29133858, 0.35826772, 0.38976378,
       0.51181102, 0.27165354, 0.11023622, 0.51181102, 0.11811024,
       0.77165354, 0.81102362, 0.20866142, 0.12992126, 0.16141732])
```

[ ]:
```
print("Median precipitation on rainy days:  ", np.median(inches[rainy]))
```

```
Median precipitation on rainy days:   0.19488188976377951
```

[ ]:
```
print("Median precipitation on summer days:  ", np.median(inches[summer]))
```

```
Median precipitation on summer days:   0.0
```

[ ]:
```
np.sort(inches[summer]) # inches[summer] sorted; now we can see why the median␣
↪is 0.0 (median is the middle value)
```

[ ]:
```
array([0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.01181102, 0.01181102, 0.01968504, 0.01968504, 0.01968504,
       0.03937008, 0.0511811 , 0.07086614, 0.09055118, 0.11811024,
       0.33070866, 0.5       , 0.75984252, 0.8503937 ])
```

[ ]:
```
print("Maximum precipitation on summer days:  ", np.max(inches[summer]))
```

```
Maximum precipitation on summer days:   0.8503937007874016
```

```
[ ]: print("Median precipitation on non-summer days:  ", np.median(inches[rainy &
     ↪~summer]))
```

```
Median precipitation on non-summer days:   0.20078740157480315
```

2.10 Fancy Indexing

Fancy indexing is just like simple indexing we have already seen, but we pass arrays of indices in place of single scalars. This allows for quick access and modification of complicated subsets of an array's values.

```
[ ]: import numpy as np

     rand = np.random.RandomState(42)
     x = rand.randint(100, size=10)
     x
```

```
[ ]: array([51, 92, 14, 71, 60, 20, 82, 86, 74, 74])
```

```
[ ]: # access three different elements
     [x[1], x[3], x[0]]
```

```
[ ]: [92, 71, 51]
```

```
[ ]: # alternatively, we can pass a single list or array of indices
     indices = [1, 3, 0]
     x[indices]
```

```
[ ]: array([92, 71, 51])
```

The result reflects the shape of the index array, not the array being indexed.

```
[ ]: x
```

```
[ ]: array([51, 92, 14, 71, 60, 20, 82, 86, 74, 74])
```

```
[ ]: indices = np.array([[2, 7],
                         [8, 4]])
     indices
```

```
[ ]: array([[2, 7],
            [8, 4]])
```

```
[ ]: # result reflects the shape of the index array
     x[indices]
```

```
[ ]: array([[14, 86],
            [74, 60]])
```

Fancy indexing in multiple dimensions

[ ]:
```
X = np.arange(12).reshape((3, 4))
X
```

[ ]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

[ ]:
```
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
X[row, col] # Result = [X[0,2], X[1,1], X[2,3]]
```

[ ]:
```
array([ 2,  5, 11])
```

The pairing of indices in fancy indexing follows all the broadcasting rules.

[ ]:
```
X
```

[ ]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

[ ]:
```
X[row[:, np.newaxis], col] # broadcasting here
```

[ ]:
```
array([[ 2,  1,  3],
       [ 6,  5,  7],
       [10,  9, 11]])
```

Combined Indexing

Fancy indexing can be combined with other indexing schemes that we are familiar with.

[ ]:
```
X
```

[ ]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

[ ]:
```
# fancy and normal indexing combined
X[2, [2, 0, 1]] # X[row_index (normal indexing), col_indices (fancy indexing)]
```

[ ]:
```
array([10,  8,  9])
```

[ ]:
```
# fancy indexing with slicing combined
X[1:, [3, 1, 2]] # X[rows_1_2, cols_3_1_2]
```

[ ]:
```
array([[ 7,  5,  6],
       [11,  9, 10]])
```

[ ]:
```
row
```

[ ]:
```
array([0, 1, 2])
```

[ ]:
```
X
```

[ ]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

[ ]:
```
# fancy indexing with masking combined
mask = np.array([1, 0, 1, 0], dtype=bool)
X[row[:, np.newaxis], mask] # skip the second and fourth col (masked)
```

[ ]:
```
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
```

2.11 Example: Selecting Random Points

A common use of fancy indexing is the selection of subsets of rows from a matrix.

Suppose, we have a NxD matrix representing N points in D dimensions.

Let's create a matrix in which the elements are drawn from a 2-dimensional normal distribution.

[ ]:
```
import numpy as np
rand = np.random.RandomState(42)

mean = [0, 0]
# covariance matrix
cov = [[1, -3],
       [-3, 5]]

X = rand.multivariate_normal(mean, cov, 100)
X.shape
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:9: RuntimeWarning:
covariance is not positive-semidefinite.
  if __name__ == '__main__':
```

[ ]:
```
(100, 2)
```

[ ]:
```
X[:5]
```

[ ]:
```
array([[-0.1382643 ,  1.11068661],
       [ 1.52302986,  1.4482756 ],
       [-0.23413696, -0.52358286],
       [ 0.76743473,  3.53122721],
```

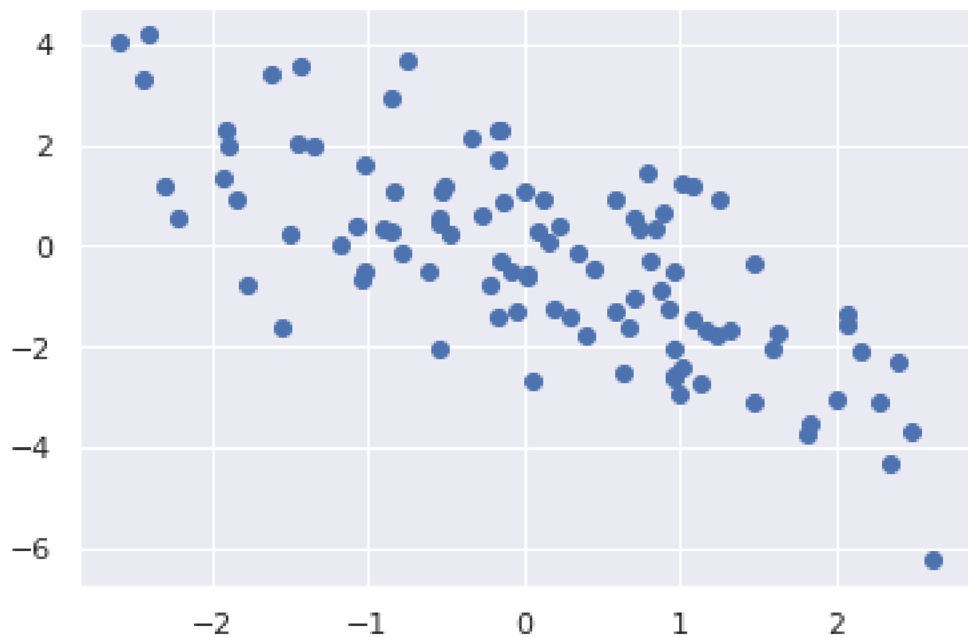
```
        [ 0.54256004, -1.04977664]])
```

[ ]:
```
X.mean(axis=0)
```

[ ]:
```
array([ 0.11680625, -0.27436292])
```

Let's visualize the points.

[ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # for plot styling

plt.scatter(X[:, 0], X[:, 1]);
```



Let's use fancy indexing to select 20 random points, by choosing 20 random indices with no repeats, and use these indices to select a portion of the original array.

[ ]:
```
indices = np.random.choice(X.shape[0], 20, replace=False)
indices
```

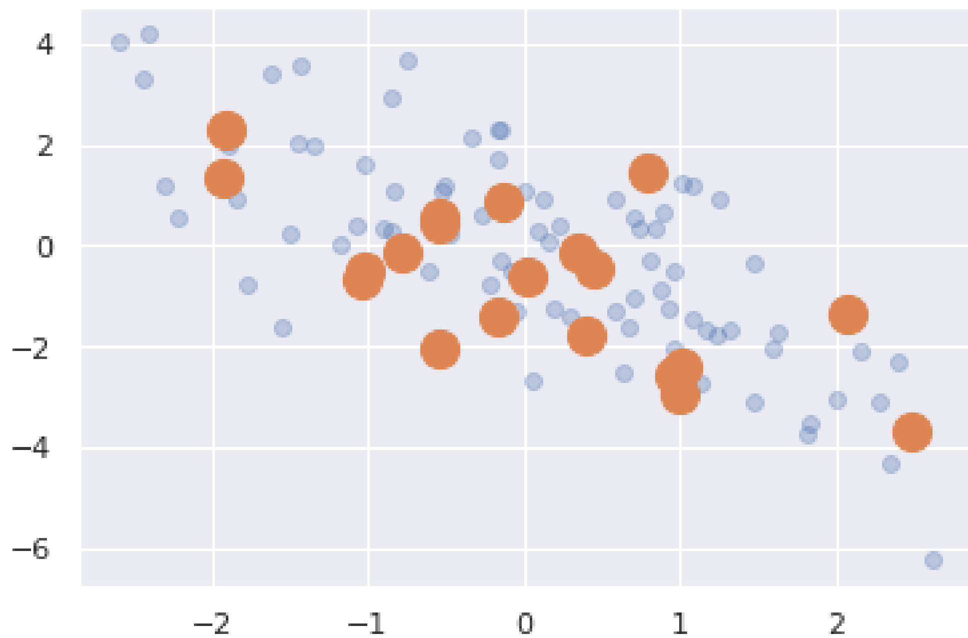
[ ]:
```
array([59,  7, 25, 21, 23, 75,  8, 15, 66,  2, 79, 24, 26, 94, 40, 48, 88,
       32, 36, 45])
```

[ ]:
```
selection = X[indices] # fancy indexing
selection.shape
```

[ ]: (20, 2)

Now let's see which points were selected by over-plotting large circles at the location of selected points.

```
[ ]: plt.scatter(X[:,0], X[:, 1], alpha=0.3)
     plt.scatter(selection[:, 0], selection[:, 1], s=200);
```



This technique can be used to quickly partition datasets, such as splitting data into train/test validation datasets.

Modifying Values with Fancy Indexing

Fancy indexing can be used to modify parts of an array.

```
[ ]: x = np.arange(10)
     i = np.array([2, 1, 8, 4])

     print(x)
     x[i] = 99
     print(x)
```

```
[0 1 2 3 4 5 6 7 8 9]
[ 0 99 99  3 99  5  6  7 99  9]
```

```
[ ]: # any assignment operator works
     print(x)
```

```
x[i] -= 10
print(x)
```

```
[ 0 99 99  3 99  5  6  7 99  9]
[ 0 89 89  3 89  5  6  7 89  9]
```

2.12 Sorting Arrays

Numpy has sorting functions which are much faster than built-in sorting functions.

In order to return the sorted version of the array without modifying the input, we can use np.sort function.

```
[ ]: x = np.array([3, 4, -3, 0, 8])

     np.sort(x) # returns a new sorted array
```

```
[ ]: array([-3,  0,  3,  4,  8])
```

```
[ ]: x # original array unchanged
```

```
[ ]: array([ 3,  4, -3,  0,  8])
```

If you want to sort the array in-place, you can use the sort method of arrays.

```
[ ]: x.sort()
     x # original array sorted in-place
```

```
[ ]: array([-3,  0,  3,  4,  8])
```

argsort: returns the *indices* of the sorted elements.

```
[ ]: x = np.array([2, 3, -2, 7, 0])
     np.argsort(x) # returns the indices of sorted elements
```

```
[ ]: array([2, 4, 0, 1, 3])
```

In order to get the elements in sorted order from the indices above, we can use fancy indexing.

```
[ ]: x[np.argsort(x)] # fancy indexing; equivalent to x[[2, 4, 0, 1, 3]]
```

```
[ ]: array([-2,  0,  2,  3,  7])
```

Sorting along rows or columns

Numpy's sorting algorithms can sort along specific rows or columns of a multi-dimensional array using the `axis` argument.

```
[ ]: rand = np.random.RandomState(42)
     X = rand.randint(0, 10, (4, 6))
```

```
X
```

```
[ ]: array([[6, 3, 7, 4, 6, 9],
            [2, 6, 7, 4, 3, 7],
            [7, 2, 5, 4, 1, 7],
            [5, 1, 4, 0, 9, 5]])
```

```
[ ]: # sort each column of X
     np.sort(X, axis=0)
```

```
[ ]: array([[2, 1, 4, 0, 1, 5],
            [5, 2, 5, 4, 3, 7],
            [6, 3, 7, 4, 6, 7],
            [7, 6, 7, 4, 9, 9]])
```

```
[ ]: # sort each row of X
     np.sort(X, axis=1)
```

```
[ ]: array([[3, 4, 6, 6, 7, 9],
            [2, 3, 4, 6, 7, 7],
            [1, 2, 4, 5, 7, 7],
            [0, 1, 4, 5, 5, 9]])
```

Please remember that the above sorts along the row or column treat each row or column as independent arrays, and any relationship between the row or column will be lost.

Partial Sorts: Partitioning

Sometimes, we do not want to sort the entire array, rather we want to simply find the K smallest values in the array. In order to do so, NumPy provide the `np.partition` function.

```
[ ]: import numpy as np
     x = np.array([7, 2, 3, 1, 6, 5, 4])
     print(x)
     np.partition(x, 3) # The first 3 elements in the returned arrays are the␣
      ↪smallest 3
```

```
[7 2 3 1 6 5 4]
```

```
[ ]: array([2, 1, 3, 4, 6, 5, 7])
```

```
[ ]: # get the largest k numbers
     -np.partition(-x, 3) # k = 3
```

```
[ ]: array([6, 7, 5, 4, 2, 3, 1])
```

We can partition along an arbitrary axis of a multidimensional array.

[ ]:
```
rand = np.random.RandomState(42)
X = rand.randint(0, 10, (4,6))
X
```

[ ]:
```
array([[6, 3, 7, 4, 6, 9],
       [2, 6, 7, 4, 3, 7],
       [7, 2, 5, 4, 1, 7],
       [5, 1, 4, 0, 9, 5]])
```

[ ]:
```
np.partition(X, 3, axis=1) # 3 smallest values along rows
```

[ ]:
```
array([[3, 4, 6, 6, 7, 9],
       [2, 3, 4, 6, 7, 7],
       [2, 1, 4, 5, 7, 7],
       [0, 4, 1, 5, 9, 5]])
```

[ ]:
```
np.partition(X, 3, axis=0) # 3 smallest values along cols
```

[ ]:
```
array([[2, 2, 5, 0, 1, 5],
       [5, 1, 4, 4, 3, 7],
       [6, 3, 7, 4, 6, 7],
       [7, 6, 7, 4, 9, 9]])
```

2.13 Structured Data: Numpy's Structured Arrays

Let's imagine that we have several categories of data on a number of people.

[ ]:
```
name = ["Alice", "Bob", "Cathy", "Steve"]
age = [35, 42, 25, 15]
weight = [60.3, 80.2, 75.1, 51.0]
```

In the above way, there is nothing that tells us that the 3 data are related.

So we need a single structure to store all those data.

NumPy can handle this through structured arrays, which are arrays with compound data types.

[ ]:
```
import numpy as np
# use a compound data type for structured arrays
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                          'formats':('U10', 'i4', 'f8')})
data.dtype
```

[ ]:
```
dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

[ ]:
```
data
```

[ ]:
```
array([('', 0, 0.), ('', 0, 0.), ('', 0, 0.), ('', 0, 0.)],
      dtype=[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

In the above code,

- U10: Unicode string of max length 10
- i4: 4-byte integer
- f8: 8-byte float

We just created an empty container array. Now let's fill in the data.

[ ]:
```
# we use the arrays we defined above to fill in the values
data['name'] = name
data['age'] = age
data['weight'] = weight

data
```

[ ]:
```
array([('Alice', 35, 60.3), ('Bob', 42, 80.2), ('Cathy', 25, 75.1),
       ('Steve', 15, 51. )],
      dtype=[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

[ ]:
```
# get all names
data['name']
```

[ ]:
```
array(['Alice', 'Bob', 'Cathy', 'Steve'], dtype='<U10')
```

[ ]:
```
# get first row of data
data[0]
```

[ ]:
```
('Alice', 35, 60.3)
```

[ ]:
```
# updating data value
data[0]['name'] = 'Micheal'
data[0]
```

[ ]:
```
('Micheal', 35, 60.3)
```

[ ]:
```
# get the name from the last row
data[-1]['name']
```

[ ]:
```
'Steve'
```

By using boolean masking, we can perform more sophisticated operations, such as filtering on age.

[ ]:
```
data[data['age'] < 30]['name']
```

[ ]:
```
array(['Cathy', 'Steve'], dtype='<U10')
```

Creating Structured Arrays

Structured array data types can be specified in a number of ways.

We saw the dictionary method above.

```
[ ]: np.dtype({'names':('name', 'age', 'weight'),
               'formats':('U10', 'i4', 'f8')})
```

```
[ ]: dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

```
[ ]: # For clarity, numerical types can be specified with Python types of NumPy␣
      ↪dtypes
     np.dtype({'names':('name', 'age', 'weight'),
               'formats':((np.str_, 10), int, np.float32)}) # Python type (int) and␣
      ↪NumPy types (np.str_, np.float32)
```

```
[ ]: dtype([('name', '<U10'), ('age', '<i8'), ('weight', '<f4')])
```

```
[ ]: # compound type can also be specified as a list of tuples
     np.dtype([('name','S10'),('age','i4'), ('weight','f8')])
```

```
[ ]: dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])
```

```
[ ]: # if name of the types do not matter, can specify the types only
     np.dtype('S10, i4, f8')
```

```
[ ]: dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

RecordArrays: Structured Arrays with a Twist

NumPy has `np.recarray` class, which is the same as structured arrays seen above, but with one additional feature: fields can be accessed as attributes with the dot notation.

```
[ ]: data['age']
```

```
[ ]: array([35, 42, 25, 15], dtype=int32)
```

```
[ ]: data_rec = data.view(np.recarray) # RecordArray
     data_rec.weight # recordarray makes it possible to access 'age' as an attribute
```

```
[ ]: array([60.3, 80.2, 75.1, 51. ])
```

But it is slower.

```
[ ]: %timeit data['age']
     %timeit data_rec['age']
     %timeit data_rec.age
```

```
The slowest run took 69.68 times longer than the fastest. This could mean that
an intermediate result is being cached.
```

```
1000000 loops, best of 5: 218 ns per loop
The slowest run took 11.11 times longer than the fastest. This could mean that
an intermediate result is being cached.
100000 loops, best of 5: 2.99 µs per loop
The slowest run took 8.51 times longer than the fastest. This could mean that an
intermediate result is being cached.
100000 loops, best of 5: 3.89 µs per loop
```

As you can see above, accessing data using RecordArray is much slower.

3 Data Manipulation with Pandas

Pandas is a package built on top of NumPy, and provides an efficient implementation of a DataFrame. DataFrames are multi-dimensional arrays with attached row and column labels, often with heterogeneous types and/or missing data.

Pandas, and it's `Series` and `DataFrame` objects, builds on the NumPy array structure and provides efficient access to data cleaning tasks that occupy much of a data scientist's time.

3.1 Pandas Objects

There are 3 fundamental Pandas data structures:

- Series
- DataFrame
- Index

3.1.1 Pandas `Series` Object

A Pandas `Series` is a one-dimensional array of indexed data.

[ ]:
```
import pandas as pd

data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

[ ]:
```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

[ ]:
```
type(data)
```

[ ]:
```
pandas.core.series.Series
```

As you can see above, the `Series` has a sequence of values as well as a sequence of indices.

[ ]:
```
data.values # accessing values of the Series
```

[ ]:
```
array([0.25, 0.5 , 0.75, 1.  ])
```

[ ]:
```
data.index # accessing indices of the Series
```

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

[ ]:
```
data[2] # accessing an element in the Series using index
```

[ ]:
```
0.75
```

[ ]:
```
data[1:3]
```

[ ]:
```
1    0.50
2    0.75
dtype: float64
```

Series as generalized NumPy Array

While `Series` looks just like regular NumPy array, the essential difference is that

- NumPy array has *implicitly* defined integer indices
- Pandas Series has *explicitly* defined integer indices associated with the values

The explicit index gives the Series object additional capabilites. For instance, the index does not need to be an integer, but can be values of any data type.

For example, we can use string as index in Series.

[ ]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
data
```

[ ]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

[ ]:
```
data['c'] # accessing value using string index
```

[ ]:
```
0.75
```

Series as specialized dictionary

Since you can use any data type as index, Pandas Series can be thought of as a specialized dictionary.

Let's construct a Series object directly from a Python dictionary.

[ ]:
```
population_dict = {'California': 38,
                   'Texas': 26,
                   'New York': 20,
                   'Florida':19,
```

```
                   'Illinois': 13
                   } # population in million

population = pd.Series(population_dict)

population
```

[ ]:
```
California    38
Texas         26
New York      20
Florida       19
Illinois      13
dtype: int64
```

[ ]:
```
population['Illinois'] # accessing value using index
```

[ ]:
```
13
```

[ ]:
```
population['California':'New York'] # note that in Series the value associated␣
 ↪with end index is also provided in the output
```

[ ]:
```
California    38
Texas         26
New York      20
dtype: int64
```

Constructing Series Objects

Here are some example of constructing Series objects.

[ ]:
```
pd.Series([1, 2, 3])
```

[ ]:
```
0    1
1    2
2    3
dtype: int64
```

[ ]:
```
# data can be a scalar, which is repeated to fill the specified index
pd.Series(7, index=[10, 20, 30, 30]) # note that duplicate indices are allowed
```

[ ]:
```
10    7
20    7
30    7
30    7
dtype: int64
```

[ ]:
```
# data can be a dictionary, in which index defaults to dictionary keys
pd.Series({2:'a', 5:'b', 3:'c'})
```

```
[ ]: 2    a
     5    b
     3    c
     dtype: object
```

```
[ ]: # index can be explicitly set if preferred
     pd.Series({2:'a', 5:'b', 3:'c'}, index=[3, 2]) # index 5 is discarded since it␣
      ↪is not in the index list
```

```
[ ]: 3    c
     2    a
     dtype: object
```

3.1.2 Pandas DataFrame Object

The DataFrame can be considered as

- generalization of a NumPy array, or
- specialization of a Python dictionary

DataFrame as a generalized NumPy Array

Just like `Series` is an analog of a one dimensional array with flexible indices,

a `DataFrame` is an analog of two-dimensional array with both flexible row indices and flexible column indices.

```
[ ]: area_dict = {'California': 423,
                  'Texas': 695,
                  'New York': 141,
                  'Florida': 170,
                  'Illinois': 150
                  } # area in thousand sq. miles
```

```
[ ]: area = pd.Series(area_dict)
     area
```

```
[ ]: California    423
     Texas         695
     New York      141
     Florida       170
     Illinois      150
     dtype: int64
```

We can use the Series `area` and `population` (we defined above) to create a single two-dimensional object.

```
[ ]: states = pd.DataFrame({'population': population,
                            'area': area
                           })
```

```
states
```

```
[ ]:            population  area
     California          38   423
     Texas               26   695
     New York            20   141
     Florida             19   170
     Illinois            13   150
```

```
[ ]: type(states)
```

```
[ ]: pandas.core.frame.DataFrame
```

As you can see, `states` is a DataFrame.

Like the Series object, DataFrame has an index attribute that gives access to the index labels.

```
[ ]: states.index # shows indices
```

```
[ ]: Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'],
     dtype='object')
```

Moreover, the DataFrame has a `columns` attribute, which is an `Index` object holding the column labels.

```
[ ]: states.columns # shows columns of the dataframe
```

```
[ ]: Index(['population', 'area'], dtype='object')
```

Thus, the DataFrame can be thought of as a generalization of a 2-dimensional NumPy array, where both the rows and columns have a generalized index for accessing the data.

DataFrame as specialized dictionary

Just as a dictionary maps a key to a value, `DataFrame` maps a column name to a `Series` of column data.

```
[ ]: states['area'] # maps a column name to a Series of column data
```

```
[ ]: California    423
     Texas         695
     New York      141
     Florida       170
     Illinois      150
     Name: area, dtype: int64
```

Constructing DataFrame objects

A DataFrame can be created in a number of ways.

[ ]:
```python
# single column DataFrame can be created from a single Series
pd.DataFrame(population, columns=['population'])
```

[ ]:
```
            population
California          38
Texas               26
New York            20
Florida             19
Illinois            13
```

[ ]:
```python
# from a list of dicts
data = [{'a': i, 'b': 2*i} for i in range(6)]
pd.DataFrame(data)
```

[ ]:
```
   a   b
0  0   0
1  1   2
2  2   4
3  3   6
4  4   8
5  5  10
```

[ ]:
```python
# Even if some keys in the dict are missing, Pandas fills them up with NaN (Not␣
↪a Number)
pd.DataFrame([{'a': 2, 'b': 5}, {'b': 7, 'c': 1}])
```

[ ]:
```
     a  b    c
0  2.0  5  NaN
1  NaN  7  1.0
```

[ ]:
```python
# from a dict of series objects
pd.DataFrame({'population': population,
              'area': area
              })
```

[ ]:
```
            population  area
California          38   423
Texas               26   695
New York            20   141
Florida             19   170
Illinois            13   150
```

[ ]:
```python
# from a 2-dimensional numpy array
import numpy as np
pd.DataFrame(np.random.rand(3, 2),
             columns = ['foo', 'bar'],
             index = ['a', 'b', 'c'])
```

[ ]:
```
        foo       bar
a  0.862197  0.533838
b  0.369716  0.627426
c  0.451185  0.119035
```

[ ]:
```
# from a numpy structured array
pd.DataFrame(np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')]))
```

[ ]:
```
   A    B
0  0  0.0
1  0  0.0
2  0  0.0
```

3.1.3 Pandas Index Object

The Pandas Index object can be thought of as an immutable array.

Let's construct an Index object from a list of integers.

[ ]:
```
import pandas as pd
ind = pd.Index([1, 5, 3, 9, 7])
ind
```

[ ]:
```
Int64Index([1, 5, 3, 9, 7], dtype='int64')
```

Index as immutable array

The Index object operates like an array in many ways.

[ ]:
```
# can use indexing notation to get values
ind[3]
```

[ ]:
```
9
```

[ ]:
```
ind[::-1] # slicing also works
```

[ ]:
```
Int64Index([7, 9, 3, 5, 1], dtype='int64')
```

[ ]:
```
# Index objects have many of the same attributes as arrays
print(ind.ndim, ind.shape, ind.size, ind.dtype)
```

```
1 (5,) 5 int64
```

The only difference between Index objects and Numpy arrays are that Index objects are immutable.

[ ]:
```
ind[0] = 5 # error due to immutability (can not update values)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-d3f90598ff21> in <module>()
```

```
----> 1 ind[0] = 5 # error due to immutability (can not update values)

/usr/local/lib/python3.7/dist-packages/pandas/core/indexes/base.py in␣
↪__setitem__(self, key, value)
   4082
   4083     def __setitem__(self, key, value):
-> 4084         raise TypeError("Index does not support mutable operations")
   4085
   4086     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

Index as ordered set

Pandas objects are designed to facilitate operations such as joins across datasets, which depend on many aspects of set arithmetic.

Set unions, intersections, differences, and other combinations can be computed on Index objects.

[ ]:
```
indA = pd.Index([1, 3, 8, 4, 9])
indB = pd.Index([2, 3, 7, 5, 9])
```

[ ]:
```
# intersection of indA and indB
indA & indB
```

[ ]:
```
Int64Index([3, 9], dtype='int64')
```

[ ]:
```
indA.intersection(indB)
```

[ ]:
```
Int64Index([3, 9], dtype='int64')
```

[ ]:
```
# union of indA and indB
indA | indB
```

[ ]:
```
Int64Index([1, 2, 3, 4, 5, 7, 8, 9], dtype='int64')
```

[ ]:
```
indA.union(indB)
```

[ ]:
```
Int64Index([1, 2, 3, 4, 5, 7, 8, 9], dtype='int64')
```

3.2 Data Indexing and Selection

We have seen the method and tools to access, set, and modify values in NumPy Arrays, for example, indexing, slicing, masking, fancy indexing etc.

Here we will see similar means of accessing and modifying values in Pandas `DataFrame` objects.

3.2.1 Data Selection in Series

The `Series` object is in many ways like a one-dimensional NumPy Array, and in many ways like a standard Python dictionary.

Series as a Dictionary

Just like a dictionary, the `Series` object provides a mapping from a collection of keys to a collection of values.

[ ]:
```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
data
```

[ ]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

We can also use dictionary-like Python expressions and methods to examine the keys/indices and values.

[ ]:
```
'd' in data # is the key (index) 'd' in data?
```

[ ]:
```
True
```

[ ]:
```
data.keys() # get all keys
```

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

[ ]:
```
list(data.items()) # get all items
```

[ ]:
```
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

`Series` objects can be modified with a dictionary-like syntax.

Just as a dictionary can be extended by assigning a new key, a `Series` can be extended by assigning to a new index value.

[ ]:
```
data['e'] = 1.25 # new key (index) and value pair added
data
```

[ ]:
```
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

Series as a one-dimensional array

A `Series` provides array-style item selection with the same basic mechanisms as NumPy arrays, e.g. slices, masks, fancy indexing etc.‘

[ ]:
```
# slicing by explicit index
data['a':'c'] # note that the end index is included in the result (different␣
↪from implicit indexing)
```

[ ]:
```
a    0.25
b    0.50
c    0.75
dtype: float64
```

[ ]:
```
# slicing by implicit index
data[:2] # index 0 and 1
```

[ ]:
```
a    0.25
b    0.50
dtype: float64
```

[ ]:
```
# masking
data[(data > 0.4) & (data < 1.0)] # get data between 0.4 and 1.0
```

[ ]:
```
b    0.50
c    0.75
dtype: float64
```

[ ]:
```
# fancy indexing
data[['c', 'e']]
```

[ ]:
```
c    0.75
e    1.25
dtype: float64
```

Indexers: `loc` and `iloc`

Pandas provides special `indexer` attributes that explicitly expose certain indexing schemes.

The `loc` attribute allows indexing and slicing that always references the explicit index.

[ ]:
```
data = pd.Series(['x', 'y', 'z'], index=[1, 3, 5])
data
```

[ ]:
```
1    x
3    y
5    z
dtype: object
```

[ ]:
```
data[1]
```

[ ]: 'x'

[ ]:
```
data.loc[1] # using explicit indexing using loc
```

[ ]: 'x'

[ ]:
```
data.loc[1:3] # slicing using explicit index by the use of loc
```

[ ]:
```
1    x
3    y
dtype: object
```

[ ]:
```
data[1:3] # implicit indexing is used when slicing without loc
```

[ ]:
```
3    y
5    z
dtype: object
```

By contrast, `iloc` is used for referencing by implicit index.

[ ]:
```
data.iloc[1] # referencing by implicit index using iloc
```

[ ]: 'y'

[ ]:
```
# without iloc
data[1] # explicit indexing
```

[ ]: 'x'

[ ]:
```
data.iloc[1:4] # slicing using implicit index by the use of iloc
```

[ ]:
```
3    y
5    z
dtype: object
```

Explicit is always better than implicit. So it is better to use `loc` and `iloc` to make explicit which indexing is intended.

3.2.2 Data Selection in DataFrame

DataFrame acts in many ways as:

- a two-dimensional or structured array
- or, a dictionary of `Series` sharing the same index

DataFrame as a dictionary

Let's see how DataFrame is analogous to dictionary.

[ ]:
```
area = pd.Series({'California': 424,
                  'Texas': 696,
                  'New York': 141,
                  'Florida': 170,
                  'Illinois': 150
                  }) # thousand sq miles
```

[ ]:
```
population = pd.Series({'California': 38,
                  'Texas': 26,
                  'New York': 19,
                  'Florida': 20,
                  'Illinois': 129
                  }) # million
```

[ ]:
```
data = pd.DataFrame({'area': area, 'population': population}) # dataframe as␣
↪dictionary
data
```

[ ]:
```
            area  population
California   424          38
Texas        696          26
New York     141          19
Florida      170          20
Illinois     150         129
```

The individual `Series` that make up the columns of the DataFrame can be accessed using dictionary style indexing of the column name.

[ ]:
```
data['area']
```

[ ]:
```
California    424
Texas         696
New York      141
Florida       170
Illinois      150
Name: area, dtype: int64
```

[ ]:
```
# for string column names, attribute-style access can be used
data.area
```

[ ]:
```
data['population'] is data.population
```

[ ]:
```
True
```

This dictionary-style syntax can also be used to modify the object, e.g. to add a new column.

[ ]:
```
data['density'] = data['population'] / data['area'] # adding a new column␣
↪'density'
data
```

[ ]:

	area	population	density
California	424	38	0.089623
Texas	696	26	0.037356
New York	141	19	0.134752
Florida	170	20	0.117647
Illinois	150	129	0.860000

DataFrame as a 2-dimensional array

`DataFrame` can also be viewed as a two-dimensional array.

[ ]:
```
data.values
```

[ ]:
```
array([[4.24000000e+02, 3.80000000e+01, 8.96226415e-02],
       [6.96000000e+02, 2.60000000e+01, 3.73563218e-02],
       [1.41000000e+02, 1.90000000e+01, 1.34751773e-01],
       [1.70000000e+02, 2.00000000e+01, 1.17647059e-01],
       [1.50000000e+02, 1.29000000e+02, 8.60000000e-01]])
```

When we view `DataFrame` as a 2-dim array, we can do many array-like operations on the `DataFrame`. For example, we can transpose the `DataFrame`.

[ ]:
```
data.T # transposing the dataframe (rows become columns; columns become rows)
```

[ ]:

	California	Texas	New York	Florida	Illinois
area	424.000000	696.000000	141.000000	170.000000	150.00
population	38.000000	26.000000	19.000000	20.000000	129.00
density	0.089623	0.037356	0.134752	0.117647	0.86

The dictionary-style indexing of columns does not allow to simply treat it as a NumPy array. For example, passing a single index to an array accesses a row.

[ ]:
```
data.values[0] # return the first row of the array
```

[ ]:
```
array([4.24000000e+02, 3.80000000e+01, 8.96226415e-02])
```

For array-style indexing, Pandas uses `loc`, `iloc`, and `ix` indexers.

Using the `iloc` indexer, we can index the underlying array as if it is a simple NumPy array (using the implicit index), but the `DataFrame` index and columns labels are maintained in the result.

[ ]:
```
# implicit indexing
data.iloc[:3, :2] # rows 0, 1, 2 and cols 0, 1
```

```
[ ]:             area  population
     California   424          38
     Texas        696          26
     New York     141          19
```

```python
# explicit indexing
data.loc[:'New York' :'population'] # rows upto 'New York'; cols up to␣
↪'population'
```

```
[ ]:             area  population
     California   424          38
     Texas        696          26
     New York     141          19
```

In the `loc` indexer, we can combine masking and fancy indexing.

```python
# select rows with density > 0.1 (masking); cols => 'population', 'density'␣
↪(fancy indexing)
data.loc[data.density > 0.1, ['population', 'density']]
```

```
[ ]:           population   density
     New York          19  0.134752
     Florida           20  0.117647
     Illinois         129  0.860000
```

Any of the indexing conventions can be used to set or update values.

```python
data.iloc[0, 2] = 0.99 # update row 0, col 2 to a new value
data
```

```
[ ]:             area  population   density
     California   424          38  0.990000
     Texas        696          26  0.037356
     New York     141          19  0.134752
     Florida      170          20  0.117647
     Illinois     150         129  0.860000
```

While indexing refers to columns, slicing refers to rows.

```python
data
```

```
[ ]:             area  population   density
     California   424          38  0.990000
     Texas        696          26  0.037356
     New York     141          19  0.134752
     Florida      170          20  0.117647
     Illinois     150         129  0.860000
```

[ ]:
```
data['area'] # indexing refers to columns
```

[ ]:
```
California    424
Texas         696
New York      141
Florida       170
Illinois      150
Name: area, dtype: int64
```

[ ]:
```
data['Texas':'Florida'] # slicing refers to rows
```

[ ]:
```
          area  population   density
Texas      696          26  0.037356
New York   141          19  0.134752
Florida    170          20  0.117647
```

Direct masking operations are also interpreted row-wise rather than column-wise.

[ ]:
```
data[data.density > 0.5]
```

[ ]:
```
            area  population  density
California   424          38     0.99
Illinois     150         129     0.86
```

3.3 Operating on Data in Pandas

As you saw, NumPy has the ability of perform quick element-wise operations, such as arithmetic operations as well as more sophisticated operations, such as trigonometric functions, exponentials, logarithms etc. Pandas inherits much of this functionality from NumPy.

In Pandas, for unary operations like negation, and trigonometric functions, the ufuncs *preserve index and column labels* in the output. For binary operations, such as addition and multiplication, Pandas automatically *assigns indices* when passing the objects to the ufuncs.

This means that keeping the context of data and combining data from different sources - both potentially error-prone tasks with Numpy Arrays - become essentially fool-proof ones with Pandas.

3.3.1 Ufuncs: Index Preservation

Any NumPy ufuncs works on Pandas `Series` and `DataFrame` objects.

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

rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4)) # a Series object
ser
```

[ ]:
```
0    6
1    3
2    7
3    4
dtype: int64
```

[ ]:
```
df = pd.DataFrame(rng.randint(0, 10, (3, 4)), columns=['A', 'B', 'C', 'D']) # a␣
↪DataFrame object
df
```

[ ]:
```
   A  B  C  D
0  1  7  5  1
1  4  0  9  5
2  8  0  9  2
```

If we apply a NumPy ufunc on either of the objects defined above, we will get another Pandas object *with the indices preserved.*

[ ]:
```
np.exp(ser) # exponential (e^element) of each element in the Series
```

[ ]:
```
0     403.428793
1      20.085537
2    1096.633158
3      54.598150
dtype: float64
```

[ ]:
```
# a bit more complex calculation
np.sin(df * np.pi / 4) # sine of each element multipled by pi/4
```

[ ]:
```
              A         B         C         D
0  7.071068e-01 -0.707107 -0.707107  0.707107
1  1.224647e-16  0.000000  0.707107 -0.707107
2 -2.449294e-16  0.000000  0.707107  1.000000
```

3.3.2 UFuncs: Index Alignment

For binary operations on two `Series` or `DataFrame` objects, Pandas will align indices in the process of performing the operations.

- **Index alignment in Series**

Suppose we are combining two different data sources such as the following.

[ ]:
```
area = pd.Series({'Alaska': 172,
                  'Texas': 696,
                  'California': 424
                  }, name = 'area')
```

[ ]:
```
population = pd.Series({'California': 38,
                        'Texas': 26,
                        'New York': 19
                        }, name = 'area')
```

Let's compute the population density.

[ ]:
```
population / area
```

[ ]:
```
Alaska              NaN
California     0.089623
New York            NaN
Texas          0.037356
Name: area, dtype: float64
```

As you can see, we get a `NaN` (Not a Number) for rows in which either of population or area value is missing.

Another example:

[ ]:
```
A = pd.Series([2, 4, 7], index=[0, 1, 2])
B = pd.Series([8, 3, 1], index=[1, 2, 3])

A + B
```

[ ]:
```
0     NaN
1    12.0
2    10.0
3     NaN
dtype: float64
```

If we do not want NaN, then we can modify the fill value using appropriate object methods in place of operators.

For example, calling `A.add(B)` is equivalent to `A + B`, but allows optional explicit specification of the fill value to replace the missing values with.

[ ]:
```
A.add(B, fill_value=0) # fill with 0 in place of NaN entries
```

[ ]:
```
0     2.0
1    12.0
2    10.0
3     1.0
dtype: float64
```

As you can see above, in the first row (index 0) the value of B is missing which is replaced by zero, and the result is `2 + 0 = 2`.

- **Index alignment in DataFrame**

A similar type of alignment takes place for both columns and indices, when performing operations on `DataFrames`.

[ ]:
```
A =  pd.DataFrame(rng.randint(0, 20, (2, 2)), columns=list('AB'))
A
```

[ ]:
```
    A   B
0  11  19
1   2   4
```

[ ]:
```
B =  pd.DataFrame(rng.randint(0, 10, (3, 3)), columns=list('BAC'))
B
```

[ ]:
```
   B  A  C
0  2  6  4
1  8  6  1
2  3  8  1
```

[ ]:
```
A + B # element-wise addition with NaN for missing values
```

[ ]:
```
      A     B   C
0  17.0  21.0 NaN
1   8.0  12.0 NaN
2   NaN   NaN NaN
```

As can be seen above, the indices and columns are aligned properly, and the indices are sorted.

Just like with the `Series`, we can use the associated object's arithmetic method and pass any desired `fill_value` to be used in place of missing entries.

[ ]:
```
A.add(B, fill_value=0)
```

[ ]:
```
      A     B    C
0  17.0  21.0  4.0
1   8.0  12.0  1.0
2   8.0   3.0  1.0
```

3.3.3 Ufuncs: Operations Between DataFrame and Series

When performing operations between a `DataFrame` and a `Series`, the index and column alignments are similarly maintained.

Operations between a `DataFrame` and a `Series` are similar to operations between a two-dimensional and one-dimensional NumPy array.

[ ]:
```
A = rng.randint(10, size=(3, 4)) # 2-dim array
A
```

```
[ ]: array([[9, 8, 9, 4],
            [1, 3, 6, 7],
            [2, 0, 3, 1]])
```

```
[ ]: A -  A[0] # A minus the first row of A (broadcasting); 2-dim array - 1-dim array
```

```
[ ]: array([[ 0,  0,  0,  0],
            [-8, -5, -3,  3],
            [-7, -8, -6, -3]])
```

Let's see how the same operation works with `DataFrame`.

```
[ ]: # dataframe
     df = pd.DataFrame(A, columns=list('QRST'))
     df
```

```
[ ]:    Q  R  S  T
     0  9  8  9  4
     1  1  3  6  7
     2  2  0  3  1
```

```
[ ]: # dataframe - series
     df - df.iloc[0]
```

```
[ ]:    Q  R  S  T
     0  0  0  0  0
     1 -8 -5 -3  3
     2 -7 -8 -6 -3
```

If you want column-wise operation, you can use the object methods with `axis` specified.

```
[ ]: df.subtract(df['S'], axis = 0) # column-wise operationm
```

```
[ ]:    Q  R  S  T
     0  0 -1  0 -5
     1 -5 -3  0  1
     2 -1 -3  0 -2
```

The preservation and alignment of indices and columns means that operations on data in Pandas will always maintain the data context.

3.4 Handling Missing Data

Real world data is rarely clean and homogeneous. Many of real world data have some amount of missing values.

Missing Data in Pandas

Pandas uses sentinels for missing data, and uses two Python null values:

- the special floating-point NaN value
- the Python `None` object

- **None: Pythonic missing data**

The first sentinel value Pandas uses is `None`. As `None` is a Python object, it can not be used in any arbitrary NumPy/Pandas array, but only in arrays with data type 'object' - that is arrays of Python objects.

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

vals1 = np.array([1, None, 3, 4])
vals1
```

[ ]:
```
array([1, None, 3, 4], dtype=object)
```

- **NaN: Missing numerical data**

The other missing data representation is NaN, which is a special floating point value.

[ ]:
```
vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype
```

[ ]:
```
dtype('float64')
```

`NaN` is like a data virus, which infects any other object it interacts with. Regardless of the operation, the result of arithmetic with `NaN` will be another `NaN`.

[ ]:
```
1 + np.nan
```

[ ]:
```
nan
```

[ ]:
```
0 * np.nan
```

[ ]:
```
nan
```

[ ]:
```
# aggregates over values involving NaN
vals2.sum(), vals2.min(), vals2.max()
```

[ ]:
```
(nan, nan, nan)
```

NumPy provides some special aggregations that ignore missing values.

[ ]:
```
np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```

[ ]:
```
(8.0, 1.0, 4.0)
```

3.4.1 NaN and None in Pandas

Pandas can handle NaN and None nearly interchangeably, converting between them where appropriate.

[ ]:
```
pd.Series([1, 2, np.nan, 4, None])
```

[ ]:
```
0    1.0
1    2.0
2    NaN
3    4.0
4    NaN
dtype: float64
```

For types that don't have available sentinel value, Pandas automatically type-casts when NA values are present.

For example, if we set a value in an integer array to np.nan, it will automatically be upcast to a floating-point type to accommodate the NA.

[ ]:
```
import pandas as pd
x = pd.Series(range(2), dtype=int)
x
```

[ ]:
```
0    0
1    1
dtype: int64
```

[ ]:
```
x[0] = None
x
```

[ ]:
```
0    NaN
1    1.0
dtype: float64
```

3.4.2 Operating on Null Values

As we saw, Pandas treats None and NaN as essentially interchangeable for indicating missing or null values. To facilitate this, there are useful methods for detecting, removing, and replacing null values in Pandas, such as: * isnull(): Generates a Boolean mask indicating missing values * notnull(): Opposite of isnull() * dropna(): Returns a filtered version of the data * fillna(): Returns a copy of the data with missing values filled or imputed

- **Detecting null values**

[ ]:
```
import pandas as pd
import numpy as np
data = pd.Series([1, np.nan, 'hello', None])
```

```
data
```

```
[ ]: 0        1
     1      NaN
     2    hello
     3     None
     dtype: object
```

```
[ ]: data.isnull() # returns a Boolean mask over the data
```

```
[ ]: 0    False
     1     True
     2    False
     3     True
     dtype: bool
```

```
[ ]: data[data.notnull()] # Boolean mask returned by notnull() in Series index
```

```
[ ]: 0        1
     2    hello
     dtype: object
```

The `isnull()` and `notnull()` methods produce similar Boolean results for DataFrames.

- **Dropping null values**

```
[ ]: data.dropna() # removes the NA values
```

```
[ ]: 0        1
     2    hello
     dtype: object
```

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

     df = pd.DataFrame([[1,       np.nan,   2],
                        [2,       3,        5],
                        [np.nan,  4,        6]])

     df
```

```
[ ]:      0    1  2
     0  1.0  NaN  2
     1  2.0  3.0  5
     2  NaN  4.0  6
```

```
[ ]: df.dropna() # drops all rows with NA
```

```
[ ]:      0    1  2
     1  2.0  3.0  5
```

```
[ ]: df.dropna(axis='columns') # same as df.dropna(axis=1); drops all columns with␣
      ↪null values
```

```
[ ]:    2
     0  2
     1  5
     2  6
```

```
[ ]: df[3] = np.nan
     df
```

```
[ ]:      0    1  2   3
     0  1.0  NaN  2 NaN
     1  2.0  3.0  5 NaN
     2  NaN  4.0  6 NaN
```

```
[ ]: df.dropna(axis='columns', how='all') # drops columns with all null values
```

```
[ ]:      0    1  2
     0  1.0  NaN  2
     1  2.0  3.0  5
     2  NaN  4.0  6
```

```
[ ]: df
```

```
[ ]:      0    1  2   3
     0  1.0  NaN  2 NaN
     1  2.0  3.0  5 NaN
     2  NaN  4.0  6 NaN
```

```
[ ]: df.dropna(axis='rows', thresh=3) # keep only rows with a minimum of 3 non-null␣
      ↪values
```

```
[ ]:      0    1  2   3
     1  2.0  3.0  5 NaN
```

- **Filling null values**

Pandas provides the `fillna()` method that returns a copy of the array with the null values replaced.

```
[ ]: data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
     data
```

```
[ ]: a    1.0
     b    NaN
```

```
c    2.0
d    NaN
e    3.0
dtype: float64
```

[ ]:
```
# fill NA entries with a single value, e.g. zero
data.fillna(0)
```

[ ]:
```
a    1.0
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

[ ]:
```
# specify a forward-fill to propagate the previous value forward
data.fillna(method='ffill')
```

[ ]:
```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

[ ]:
```
# specify a back-fill to propagate the next values forward
data.fillna(method='bfill')
```

[ ]:
```
a    1.0
b    2.0
c    2.0
d    3.0
e    3.0
dtype: float64
```

For `DataFrames`, the options are similar, but we can also specify the `axis` along which the fills take place.

[ ]:
```
df
```

[ ]:
```
     0    1  2   3
0  1.0  NaN  2 NaN
1  2.0  3.0  5 NaN
2  NaN  4.0  6 NaN
```

[ ]:
```
df.fillna(method='ffill', axis=1)
```

```
[ ]:      0    1    2    3
     0  1.0  1.0  2.0  2.0
     1  2.0  3.0  5.0  5.0
     2  NaN  4.0  6.0  6.0
```

3.5 Combining Datasets: Merge and Join

Pandas has high-performance in-memory join and merge operations.

3.5.1 Relational Algebra

The behaviour implemented in `pd.merge()` is a subset of what is known as *relational algebra*, which forms the foundation of operations in most databases.

3.5.2 Categories of Joins

The `pd.merge()` function implements a number of types of joins:

- one-to-one
- many-to-one
- many-to-many

- **One-to-one joins**

```
[ ]: import pandas as pd

     df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                         'group':['Accounting', 'Engineering', 'Engineering', 'HR']})

     df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                         'hire_date':[2004, 2008, 2011, 2009]})

     print(df1); print('-'*20); print(df2)
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
--------------------
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2011
3      Sue       2009
```

To combine the above two dataframes, we can use the `pd.merge()` function.

```
[ ]: df3 = pd.merge(df1, df2)
     df3
```

```
[ ]:   employee        group  hire_date
     0      Bob   Accounting       2008
     1     Jake  Engineering       2011
     2     Lisa  Engineering       2004
     3      Sue           HR       2009
```

- **Many-to-one joins**

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. The resulting DataFrame will preserve those duplicate entries as appropriate.

```
[ ]: df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'], 'supervisor':␣
      ↪['Carly', 'Guido', 'Steve']})
     print(df3); print('-'*20); print(df4)
```

```
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2011
2     Lisa  Engineering       2004
3      Sue           HR       2009
--------------------
         group supervisor
0   Accounting      Carly
1  Engineering      Guido
2           HR      Steve
```

```
[ ]: pd.merge(df3, df4)
```

```
[ ]:   employee        group  hire_date supervisor
     0      Bob   Accounting       2008      Carly
     1     Jake  Engineering       2011      Guido
     2     Lisa  Engineering       2004      Guido
     3      Sue           HR       2009      Steve
```

```
[ ]: pd.merge(df4, df3)
```

```
[ ]:          group supervisor employee  hire_date
     0   Accounting      Carly      Bob       2008
     1  Engineering      Guido     Jake       2011
     2  Engineering      Guido     Lisa       2004
     3           HR      Steve      Sue       2009
```

- **Many-to-many joins**

If the key column in both the left and right array contains duplicates, the the result is a many-to-many merge.

```
[ ]: df5 = pd.DataFrame({'group': ['Accounting', 'Accounting', 'Engineering',␣
      ↪'Engineering', 'HR', 'HR'],
```

```
                    'skills': ['math', 'spreadsheets', 'coding', 'linux',␣
↪'spreadsheets', 'organization']})

print(df1); print('-'*20); print(df5)
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
--------------------
         group        skills
0   Accounting          math
1   Accounting  spreadsheets
2  Engineering        coding
3  Engineering         linux
4           HR  spreadsheets
5           HR  organization
```

[ ]:
```
pd.merge(df1, df5)
```

[ ]:
```
  employee        group        skills
0      Bob   Accounting          math
1      Bob   Accounting  spreadsheets
2     Jake  Engineering        coding
3     Jake  Engineering         linux
4     Lisa  Engineering        coding
5     Lisa  Engineering         linux
6      Sue           HR  spreadsheets
7      Sue           HR  organization
```

[ ]:
```
pd.merge(df5, df1)
```

[ ]:
```
         group        skills employee
0   Accounting          math      Bob
1   Accounting  spreadsheets      Bob
2  Engineering        coding     Jake
3  Engineering        coding     Lisa
4  Engineering         linux     Jake
5  Engineering         linux     Lisa
6           HR  spreadsheets      Sue
7           HR  organization      Sue
```

3.5.3 Specification of the Merge key

As we saw, `pd.merge()` looks for one or more matching column names between the two inputs, and uses this as the key.

However, the column names may not match perfectly in some cases. In such cases `pd.merge()` provides variety of options for handling this.

- **The `on` keyword**

You can explicitly specify the name of the key column using the `on` keyword, which takes a column name or a list of column names.

[ ]:
```python
print(df1); print('-'*20); print(df2)
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
--------------------
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2011
3      Sue       2009
```

[ ]:
```python
pd.merge(df1, df2, on='employee')
```

[ ]:

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2011
2	Lisa	Engineering	2004
3	Sue	HR	2009

This option works only if both the left and the right DataFrames have the specified column name.

- **The `left_on` and `right_on` keywords**

Sometimes you may want to merge two DataFrames on column name/s with different names in the right and the left DataFrame.

For instance, you may have 'name' as employee name in one DF and 'employee' on the other.

Let's see how we can join on those columns.

[ ]:
```python
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]
                    })

print(df1); print('-'*20); print(df3)
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
```

```
--------------------
   name  salary
0   Bob   70000
1  Jake   80000
2  Lisa  120000
3   Sue   90000
```

[ ]:
```
# joining on 'employee' and 'name'
pd.merge(df1, df3, left_on='employee', right_on='name')
```

[ ]:
```
  employee        group  name  salary
0      Bob   Accounting   Bob   70000
1     Jake  Engineering  Jake   80000
2     Lisa  Engineering  Lisa  120000
3      Sue           HR   Sue   90000
```

We can drop the redundant column as shown below.

[ ]:
```
# 'name' column dropped
pd.merge(df1, df3, left_on='employee', right_on='name').drop('name', axis=1)
```

[ ]:
```
  employee        group  salary
0      Bob   Accounting   70000
1     Jake  Engineering   80000
2     Lisa  Engineering  120000
3      Sue           HR   90000
```

- **The `left_index` and `right_index` keywords**

You can also merge on an index rather than a column.

[ ]:
```
df1a = df1.set_index('employee') # setting the 'employee' column as explicit␣
↪index on df1
df2a = df2.set_index('employee') # setting the 'employee' column as explicit␣
↪index on df2

print(df1a); print('-'*20); print(df2a)
```

```
                group
employee
Bob        Accounting
Jake      Engineering
Lisa      Engineering
Sue                HR
--------------------
          hire_date
employee
Lisa           2004
Bob            2008
```

```
Jake             2011
Sue              2009
```

[ ]:
```python
pd.merge(df1a, df2a, left_index=True, right_index=True)
```

[ ]:
```
                group  hire_date
employee
Bob        Accounting       2008
Jake      Engineering       2011
Lisa      Engineering       2004
Sue                HR       2009
```

The `join()` method performs a merge that defaults to joining on indices.

[ ]:
```python
df1a.join(df2a)
```

[ ]:
```
                group  hire_date
employee
Bob        Accounting       2008
Jake      Engineering       2011
Lisa      Engineering       2004
Sue                HR       2009
```

3.5.4 Specifying Set Arithmetic for Joins

Sometimes there are cases where a value appears in one key column and not on the other.

[ ]:
```python
df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'beans', 'bread']},
                    columns=['name', 'food'])

df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['wine', 'beer']},
                    columns=['name', 'drink'])

print(df6); print('-'*20); print(df7);
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
--------------------
     name drink
0    Mary  wine
1  Joseph  beer
```

[ ]:
```python
pd.merge(df6, df7)
```

```
[ ]:    name   food drink
     0  Mary  bread  wine
```

In the above case, the result contains the intersection of the two sets of inputs, which is known as *inner join*.

```
[ ]: pd.merge(df6, df7, how='inner') # explicit inner join
```

```
[ ]:    name   food drink
     0  Mary  bread  wine
```

The other options are: * outer * left * right

An *outer join* returns a join over the union of the input columns, and fills in all missing values with NA's.

```
[ ]: print(df6); print('-'*20); print(df7)
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
--------------------
     name drink
0    Mary  wine
1  Joseph  beer
```

```
[ ]: pd.merge(df6, df7, how='outer')
```

```
[ ]:      name   food drink
     0   Peter   fish   NaN
     1    Paul  beans   NaN
     2    Mary  bread  wine
     3  Joseph    NaN  beer
```

The *left join* returns a join over the left entries.

```
[ ]: pd.merge(df6, df7, how='left')
```

```
[ ]:     name   food drink
     0  Peter   fish   NaN
     1   Paul  beans   NaN
     2   Mary  bread  wine
```

The *right join* returns a join over the right entries.

```
[ ]: pd.merge(df6, df7, how='right')
```

```
[ ]:      name   food drink
     0     Mary  bread  wine
     1   Joseph    NaN  beer
```

3.5.5 Overlapping Column Names: The suffixes Keyword

There may be cases where the two input DataFrames have conflicting column names.

```
[ ]: df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                         'rank': [1, 2, 3, 4]})

     df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                         'rank': [3, 1, 4, 2]})

     print(df8); print('-'*20) ;print(df9);
```

```
   name  rank
0   Bob     1
1  Jake     2
2  Lisa     3
3   Sue     4
--------------------
   name  rank
0   Bob     3
1  Jake     1
2  Lisa     4
3   Sue     2
```

```
[ ]: pd.merge(df8, df9, on="name")
```

```
[ ]:    name  rank_x  rank_y
     0   Bob       1       3
     1  Jake       2       1
     2  Lisa       3       4
     3   Sue       4       2
```

If these defaults are not appropriate, then it is possible to specify a custom suffix using the `suffixes` keyword.

```
[ ]: pd.merge(df8, df9, on='name', suffixes=['_Left', '_Right'])
```

```
[ ]:    name  rank_Left  rank_Right
     0   Bob          1           3
     1  Jake          2           1
     2  Lisa          3           4
     3   Sue          4           2
```

3.5.6 Example: US States Data

Merge and join operations are useful when combining data from different data sources. Let's consider an example of some data about US states and their population.

[ ]:
```
# download state population data
!curl -O https://raw.githubusercontent.com/jakevdp/data-USstates/master/
↪state-population.csv
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 57935  100 57935    0     0   296k      0 --:--:-- --:--:-- --:--:--  296k
```

[ ]:
```
# download state areas data
!curl -O https://raw.githubusercontent.com/jakevdp/data-USstates/master/
↪state-areas.csv
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   835  100   835    0     0   5641      0 --:--:-- --:--:-- --:--:--  5641
```

[ ]:
```
# download state abbreviations data
!curl -O https://raw.githubusercontent.com/jakevdp/data-USstates/master/
↪state-abbrevs.csv
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   872  100   872    0     0   4316      0 --:--:-- --:--:-- --:--:--  4295
```

Let's take a look at the 3 datasets.

[ ]:
```
import pandas as pd

pop = pd.read_csv('/content/state-population.csv')
areas = pd.read_csv('/content/state-areas.csv')
abbrevs = pd.read_csv('/content/state-abbrevs.csv')
```

[ ]:
```
print(pop.head()); print('-'*20); print(areas.head()); print('-'*20);␣
↪print(abbrevs.head());
```

```
  state/region     ages  year  population
0           AL  under18  2012   1117489.0
1           AL    total  2012   4817528.0
2           AL  under18  2010   1130966.0
3           AL    total  2010   4785570.0
4           AL  under18  2011   1125763.0
--------------------
        state  area (sq. mi)
0     Alabama          52423
1      Alaska         656425
```

```
2     Arizona         114006
3    Arkansas          53182
4  California         163707
--------------------
        state abbreviation
0     Alabama           AL
1      Alaska           AK
2     Arizona           AZ
3    Arkansas           AR
4  California           CA
```

Now let's try to rank the states by their 2010 population density.

We can start with many-to-one merge that will give us the full state name within the population DataFrame. We want to merge on the `state/region` column of `pop`, and `abbreviation` column of `abbrevs`

We will use `how='outer'` in order to make sure that no data is thrown away due to mismatched labels.

[ ]:
```
merged = pd.merge(pop, abbrevs, how='outer', left_on='state/region',␣
 ↪right_on='abbreviation')
merged = merged.drop('abbreviation', axis=1)
merged.head()
```

[ ]:
```
  state/region     ages  year  population    state
0           AL  under18  2012   1117489.0  Alabama
1           AL    total  2012   4817528.0  Alabama
2           AL  under18  2010   1130966.0  Alabama
3           AL    total  2010   4785570.0  Alabama
4           AL  under18  2011   1125763.0  Alabama
```

Let's double check whether there are any mismatches here by looking for row with nulls.

[ ]:
```
merged.isnull().any()
```

[ ]:
```
state/region    False
ages            False
year            False
population       True
state            True
dtype: bool
```

We can see that there are nulls in `population` column. Let's see which these are.

[ ]:
```
merged[merged['population'].isnull()].head()
```

[ ]:
```
     state/region     ages  year  population state
2448           PR  under18  1990         NaN   NaN
2449           PR    total  1990         NaN   NaN
```

```
2450           PR    total  1991         NaN   NaN
2451           PR  under18  1991         NaN   NaN
2452           PR    total  1993         NaN   NaN
```

We now know that the **population** values for 'PR' (Puerto Rico) are null.

Moreover, the **state** column values are NaN's as well, which means that there are no corresponding entries in the abbrevs key.

Let's see which regions lack this match.

[ ]:
```python
merged.loc[merged['state'].isnull(), 'state/region'].unique()
```

[ ]:
```
array(['PR', 'USA'], dtype=object)
```

We can see that the **population** data includes entries for 'PR' and 'USA', while these entries do not appear in the state abbreviation key.

Let's fix that.

[ ]:
```python

```

[ ]:
```python
merged.loc[merged['state/region'] == 'PR', 'state'] = 'Puerto Rico' # assign␣
↪'Puerto Rico' to 'state' column corresponding to 'PR' in 'state/region'
merged.loc[merged['state/region'] == 'USA', 'state'] = 'United States' # assign␣
↪'United States' to 'state' column corresponding to 'USA' in 'state/region'
merged.isnull().any()
```

[ ]:
```
state/region    False
ages            False
year            False
population       True
state           False
dtype: bool
```

There are no more nulls in **state** column.

We can merge the results with the area data.

[ ]:
```python
final = pd.merge(merged, areas, on='state', how='left')
final.head()
```

[ ]:

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

Let's check for nulls to see if there are any mismatches.

[ ]:
```
final.isnull().any()
```

[ ]:
```
state/region     False
ages             False
year             False
population        True
state            False
area (sq. mi)     True
dtype: bool
```

There are nulls in the area column. Let's see which regions they are.

[ ]:
```
final['state'][final['area (sq. mi)'].isnull()].unique() # which of the state/s␣
↪have 'area (sq. mi)' null
```

[ ]:
```
array(['United States'], dtype=object)
```

Let's drop those rows with NA's.

[ ]:
```
final.dropna(inplace=True)
final.isnull().any() # no nulls
```

[ ]:
```
state/region     False
ages             False
year             False
population       False
state            False
area (sq. mi)    False
dtype: bool
```

Now let's select the portion of data corresponding to 2010, and the total population, as we want to rank states by population density.

[ ]:
```
data2010 = final.query("year == 2010 & ages == 'total'")
data2010.head()
```

[ ]:

	state/region	ages	year	population	state	area (sq. mi)
3	AL	total	2010	4785570.0	Alabama	52423.0
91	AK	total	2010	713868.0	Alaska	656425.0
101	AZ	total	2010	6408790.0	Arizona	114006.0
189	AR	total	2010	2922280.0	Arkansas	53182.0
197	CA	total	2010	37333601.0	California	163707.0

Let's now compute the population density and display it in order.

[ ]:
```
# reindex data on the state
data2010.set_index('state', inplace=True)
```

```
# compute population density
density = data2010['population'] / data2010['area (sq. mi)']
```

[ ]:
```
density.sort_values(ascending=False, inplace=True)
density.head()
```

[ ]:
```
state
District of Columbia    8898.897059
Puerto Rico             1058.665149
New Jersey              1009.253268
Rhode Island             681.339159
Connecticut              645.600649
dtype: float64
```

We can see that DC has the highest population density in 2010.

Let's also check the end of the list.

[ ]:
```
density.tail()
```

[ ]:
```
state
South Dakota    10.583512
North Dakota     9.537565
Montana          6.736171
Wyoming          5.768079
Alaska           1.087509
dtype: float64
```

As expected Alaska has the least population density.

3.6 Aggregation and Grouping

We can do efficient summarization, such as computing aggregations (`sum()`, `mean()`, `median()` etc) in which a single number gives insight into the nature of a large dataset.

3.6.1 Planets Data

Let's use the Planets dataset, which is available on Seaborn Package. It gives information on planets that astronomers have discovered around other stars.

We can download the dataset as follows.

[ ]:
```
import seaborn as sb
planets = sb.load_dataset('planets')

planets.shape
```

[ ]:
```
(1035, 6)
```

[ ]:
```
planets.head()
```

```
[ ]:            method  number  orbital_period   mass  distance  year
     0  Radial Velocity       1         269.300   7.10     77.40  2006
     1  Radial Velocity       1         874.774   2.21     56.95  2008
     2  Radial Velocity       1         763.000   2.60     19.84  2011
     3  Radial Velocity       1         326.030  19.40    110.62  2007
     4  Radial Velocity       1         516.220  10.50    119.47  2009
```

The Planets dataset has details on more than 1000 exoplanets discovered up to 2014.

3.6.2 Simple Aggregation in Pandas

Just like in NumPy arrays, in Pandas Series the aggregates return a single value.

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

     rng = np.random.RandomState(42)

     # Series
     ser = pd.Series(rng.rand(5))

     ser
```

```
[ ]: 0    0.374540
     1    0.950714
     2    0.731994
     3    0.598658
     4    0.156019
     dtype: float64
```

```
[ ]: ser.sum()
```

```
[ ]: 2.811925491708157
```

```
[ ]: ser.mean()
```

```
[ ]: 0.5623850983416314
```

For DataFrame, by default, the aggregates return results within each column.

```
[ ]: # DataFrame
     df = pd.DataFrame({'A': rng.rand(5),
                        'B': rng.rand(5)})

     df
```

```
[ ]:           A         B
     0  0.155995  0.020584
```

```
1  0.058084  0.969910
2  0.866176  0.832443
3  0.601115  0.212339
4  0.708073  0.181825
```

[ ]:
```
df.mean() # returns means by columns by default
```

[ ]:
```
A    0.477888
B    0.443420
dtype: float64
```

[ ]:
```
# mean by rows
df.mean(axis=1)
```

[ ]:
```
0    0.088290
1    0.513997
2    0.849309
3    0.406727
4    0.444949
dtype: float64
```

[ ]:
```
#alternatively
df.mean(axis='columns')
```

[ ]:
```
0    0.088290
1    0.513997
2    0.849309
3    0.406727
4    0.444949
dtype: float64
```

There is a method `describe()` that computes several common aggregates for each column and returns the result.

[ ]:
```
planets.describe()
```

[ ]:

	number	orbital_period	mass	distance	year
count	1035.000000	992.000000	513.000000	808.000000	1035.000000
mean	1.785507	2002.917596	2.638161	264.069282	2009.070531
std	1.240976	26014.728304	3.818617	733.116493	3.972567
min	1.000000	0.090706	0.003600	1.350000	1989.000000
25%	1.000000	5.442540	0.229000	32.560000	2007.000000
50%	1.000000	39.979500	1.260000	55.250000	2010.000000
75%	2.000000	526.005000	3.040000	178.500000	2012.000000
max	7.000000	730000.000000	25.000000	8500.000000	2014.000000

[ ]:
```
planets.dropna().describe() # drop the NA's and compute the aggregates
```

[ ]:

```
       number  orbital_period        mass    distance         year
count  498.00000      498.000000  498.000000  498.000000   498.000000
mean     1.73494      835.778671    2.509320   52.068213  2007.377510
std      1.17572     1469.128259    3.636274   46.596041     4.167284
min      1.00000        1.328300    0.003600    1.350000  1989.000000
25%      1.00000       38.272250    0.212500   24.497500  2005.000000
50%      1.00000      357.000000    1.245000   39.940000  2009.000000
75%      2.00000      999.600000    2.867500   59.332500  2011.000000
max      6.00000    17337.500000   25.000000  354.000000  2014.000000
```

The next level of data summarization is the `groupby` operation which allows to quickly and efficiently compute aggregates on subsets of data.

3.6.3 GroupBy: Split, Apply, Combine

As seen in the above figure,

- The *split* step breaks up and groups a `DataFrame` depending on the value of the specified key.
- The *apply* step involves computing some function (usually an aggregate, transformation, or filtering) within the individual groups.
- The *combine* step merges the results of these individual operations into an output array.

Let's look at an example.

[ ]:

```python
import pandas as pd

df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])

df
```

[ ]:

```
  key  data
0   A     0
1   B     1
2   C     2
3   A     3
4   B     4
5   C     5
```

We can compute the most basic *split-apply-combine* operation with the `groupby()` method of DataFrames, by passing the name of the desired key column.

[ ]:

```python
df_group_by = df.groupby('key')
type(df_group_by)
```

```
[ ]: pandas.core.groupby.generic.DataFrameGroupBy
```

Note that what is returned is a `DataFrameGroupBy` object. It can be considered as a special view of the `DataFrame`, which does no actual computation until the aggregation is applied. This "lazy evaluation" approach means that aggregates can be implemented very efficiently.

```
[ ]: df_group_by.sum()
```

```
[ ]:      data
     key
     A       3
     B       5
     C       7
```

```
[ ]: df_group_by.max()
```

```
[ ]:      data
     key
     A       3
     B       4
     C       5
```

You can apply virtually any common Pandas or NumPy aggregation function like you saw above.

The GroupBy object

- **Column indexing**

The `GroupBy` object supports column indexing in the same way as the DataFrame, and returns a modified `GroupBy` object.

Example:

```
[ ]: planets
```

```
[ ]:               method  number  orbital_period   mass  distance  year
     0     Radial Velocity       1      269.300000   7.10     77.40  2006
     1     Radial Velocity       1      874.774000   2.21     56.95  2008
     2     Radial Velocity       1      763.000000   2.60     19.84  2011
     3     Radial Velocity       1      326.030000  19.40    110.62  2007
     4     Radial Velocity       1      516.220000  10.50    119.47  2009
     …                …      …               …      …        …    …
     1030          Transit       1        3.941507    NaN    172.00  2006
     1031          Transit       1        2.615864    NaN    148.00  2007
     1032          Transit       1        3.191524    NaN    174.00  2007
     1033          Transit       1        4.125083    NaN    293.00  2008
     1034          Transit       1        4.187757    NaN    260.00  2008

     [1035 rows x 6 columns]
```

```
[ ]: planets.groupby('method') # DataFrameGroupBy object
```

```
[ ]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f68e833e450>
```

```
[ ]: planets_series_groupby = planets.groupby('method')['orbital_period'] #␣
     ↪SeriesGroupBy object
     planets_series_groupby
```

```
[ ]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x7f68e7ee3f50>
```

```
[ ]: # As with GroupBy object, no computation is done until we call some aggregate␣
     ↪on the object (lazy evaluation)
     planets_series_groupby.median()
```

```
[ ]: method
     Astrometry                          631.180000
     Eclipse Timing Variations          4343.500000
     Imaging                           27500.000000
     Microlensing                       3300.000000
     Orbital Brightness Modulation         0.342887
     Pulsar Timing                        66.541900
     Pulsation Timing Variations        1170.000000
     Radial Velocity                     360.200000
     Transit                               5.714932
     Transit Timing Variations            57.011000
     Name: orbital_period, dtype: float64
```

The above result gives an idea of the general scale of orbital periods (in days) associated with each method.

- **Iteration over groups**

The `GroupBy` object supports direct iteration over the groups, returning each group as a Series or DataFrame.

```
[ ]: for (method, group) in planets.groupby('method'):
       print('{0:30s} shape={1}'.format(method, group.shape))
```

```
Astrometry                     shape=(2, 6)
Eclipse Timing Variations      shape=(9, 6)
Imaging                        shape=(38, 6)
Microlensing                   shape=(23, 6)
Orbital Brightness Modulation  shape=(3, 6)
Pulsar Timing                  shape=(5, 6)
Pulsation Timing Variations    shape=(1, 6)
Radial Velocity                shape=(553, 6)
Transit                        shape=(397, 6)
Transit Timing Variations      shape=(4, 6)
```

- **Dispatch methods**

Any method not explicitly implemented by the `GroupBy` object will be passed through and called on the groups, whether they are DataFrame or Series objects.

For example, we can use the `describe()` method of DataFrames to perform a set of aggregations that describe each group in the data.

[ ]:
```
planets.groupby('method').describe()
```

[ ]:
```
                              number            ...     year
                               count      mean  ...      75%     max
method                                          ...
Astrometry                       2.0  1.000000  ...  2012.25  2013.0
Eclipse Timing Variations        9.0  1.666667  ...  2011.00  2012.0
Imaging                         38.0  1.315789  ...  2011.00  2013.0
Microlensing                    23.0  1.173913  ...  2012.00  2013.0
Orbital Brightness Modulation    3.0  1.666667  ...  2012.00  2013.0
Pulsar Timing                    5.0  2.200000  ...  2003.00  2011.0
Pulsation Timing Variations      1.0  1.000000  ...  2007.00  2007.0
Radial Velocity                553.0  1.721519  ...  2011.00  2014.0
Transit                        397.0  1.954660  ...  2013.00  2014.0
Transit Timing Variations        4.0  2.250000  ...  2013.25  2014.0

[10 rows x 40 columns]
```

3.6.4 Aggregate, filter, transform, apply

The `GroupBy` objects have * `aggregate()` * `filter()` * `transform()` * `apply()`

methods that efficiently implement a variety of useful operations before combining the grouped data.

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

rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                  'data1': range(6),
                  'data2': rng.randint(0, 10, 6)},
                  columns = ['key', 'data1', 'data2'])

df
```

[ ]:
```
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
```

```
4   B       4       7
5   C       5       9
```

- **Aggregation**

We have seen aggregations with `sum()`, `median()`, and the like, but `aggregate()` method allows for even more flexibility.

It can take a function, or a list thereof, and compute all the aggregates at once.

[ ]:
```
df.groupby('key').aggregate([min, np.median, max])
```

[ ]:
```
    data1            data2
      min median max   min median max
key
A       0    1.5   3     3    4.0   5
B       1    2.5   4     0    3.5   7
C       2    3.5   5     3    6.0   9
```

Another useful pattern is that we can pass a dictionary mapping column names to operations to be applied on that column.

[ ]:
```
df.groupby('key').aggregate({'data1': 'min',
                             'data2': 'max'})
```

[ ]:
```
     data1  data2
key
A        0      5
B        1      7
C        2      9
```

- **Filtering**

Filtering allows to drop data based on the group properties.

For example, we want to keep all the groups that have the standard deviation larger than some critical value, then we can write the following code.

[ ]:
```
def filter_func(x):
  return x['data2'].std() > 4

print(df)
print('-'*20)
print(df.groupby('key').std())
print('-'*20)
print(df.groupby('key').filter(filter_func))
```

```
  key  data1  data2
0   A      0      5
1   B      1      0
```

```
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
--------------------
       data1     data2
key
A    2.12132  1.414214
B    2.12132  4.949747
C    2.12132  4.242641
--------------------
  key  data1  data2
1   B      1      0
2   C      2      3
4   B      4      7
5   C      5      9
```

In the above result, key A does not have standard deviation greater than 4, so it is not included (last table).

- **Transformation**

While aggregation returns a reduced version of the data, transformation can return some transformed version of the full data to recombine.

For such transformation, the output is the same shape as the input.

As example is to center the data by subtracting the group-wise mean.

[ ]:
```python
df.groupby('key').transform(lambda x: x - x.mean())
```

[ ]:
```
   data1  data2
0   -1.5    1.0
1   -1.5   -3.5
2   -1.5   -3.0
3    1.5   -1.0
4    1.5    3.5
5    1.5    3.0
```

- **The `apply()` method**

The `apply()` method lets us apply an arbitrary function to the group results. The function takes a DataFrame, and returns either a Pandas object (e.g. DataFrame, or Series) or a scalar; and then the combine operation will be tailored to the type of output returned.

Below is and example of `apply()` which normalizes the first column by the sum of the second.

[ ]:
```python
def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x
```

```
print(df)
print('-'*20)
print(df.groupby('key').apply(norm_by_data2))
```

```
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
--------------------
  key     data1  data2
0   A  0.000000      5
1   B  0.142857      0
2   C  0.166667      3
3   A  0.375000      3
4   B  0.571429      7
5   C  0.416667      9
```

`apply()` within a `GroupBy` is very flexible: the only criterion is that the function takes a DataFrame and returns a Pandas object or scalar; what you do in the middle is up to you.

Grouping Example

Let's look at an example using grouping in planets data.

[ ]:
```
planets
```

[ ]:
```
               method  number  orbital_period   mass  distance  year
0     Radial Velocity       1      269.300000   7.10     77.40  2006
1     Radial Velocity       1      874.774000   2.21     56.95  2008
2     Radial Velocity       1      763.000000   2.60     19.84  2011
3     Radial Velocity       1      326.030000  19.40    110.62  2007
4     Radial Velocity       1      516.220000  10.50    119.47  2009
…                 …     …             …     …        …     …
1030          Transit       1        3.941507    NaN    172.00  2006
1031          Transit       1        2.615864    NaN    148.00  2007
1032          Transit       1        3.191524    NaN    174.00  2007
1033          Transit       1        4.125083    NaN    293.00  2008
1034          Transit       1        4.187757    NaN    260.00  2008

[1035 rows x 6 columns]
```

[ ]:
```
decade = 10 * (planets['year'] // 10)
decade = decade.astype(str) + 's'
decade.name = 'decade'
decade
```

[ ]:
```
0       2000s
1       2000s
2       2010s
3       2000s
4       2000s
        …
1030    2000s
1031    2000s
1032    2000s
1033    2000s
1034    2000s
Name: decade, Length: 1035, dtype: object
```

[ ]:
```
planets.groupby(['method', decade])['number'].sum().unstack().fillna(0)
```

[ ]:
```
decade                         1980s  1990s  2000s  2010s
method
Astrometry                       0.0    0.0    0.0    2.0
Eclipse Timing Variations        0.0    0.0    5.0   10.0
Imaging                          0.0    0.0   29.0   21.0
Microlensing                     0.0    0.0   12.0   15.0
Orbital Brightness Modulation    0.0    0.0    0.0    5.0
Pulsar Timing                    0.0    9.0    1.0    1.0
Pulsation Timing Variations      0.0    0.0    1.0    0.0
Radial Velocity                  1.0   52.0  475.0  424.0
Transit                          0.0    0.0   64.0  712.0
Transit Timing Variations        0.0    0.0    0.0    9.0
```

[ ]:
```
planets.groupby(['method', decade])['number'].sum().fillna(0) # without␣
↪unstack()
```

[ ]:
```
method                         decade
Astrometry                     2010s       2
Eclipse Timing Variations      2000s       5
                               2010s      10
Imaging                        2000s      29
                               2010s      21
Microlensing                   2000s      12
                               2010s      15
Orbital Brightness Modulation  2010s       5
Pulsar Timing                  1990s       9
                               2000s       1
                               2010s       1
Pulsation Timing Variations    2000s       1
Radial Velocity                1980s       1
                               1990s      52
                               2000s     475
```

```
                              2010s     424
Transit                       2000s      64
                              2010s     712
Transit Timing Variations     2010s       9
Name: number, dtype: int64
```

The above example shows the power of combining many of the operations we discussed on real datasets. Here we gain an high-level understanding of when and how planets have been discovered over the past several decades.

3.7 Working with Time Series

Pandas has extensive set of tools for working with dates, times, and time-indexed data.

Date and time data comes in a few flavors. * **Time stamps**: reference particular moments in time (for example - June 14, 2009 at 6:25 PM). * **Time intervals** and **periods**: reference a length of time between a particular beginning and end point, for example - the year 2020. Periods usually reference a special case of time intervals in which each interval is of uniform length and does not overlap (e.g. 24 hour-long periods constituting days). * **Time deltas** or **durations**: reference an exact length of time (e.g. a duration of 26.65 seconds)

3.7.1 Dates and Times in Python

Python has a number of available representations of dates, times, deltas, and timestamps.

- **Native Python dates and times: datetime and dateutil**

Python's basic objects for working with dates and times reside in the built-in `datetime` module.

[ ]:
```
from datetime import datetime

# manually build a date using datetime
datetime(year=2016, month=7, day=24)
```

[ ]:
```
datetime.datetime(2016, 7, 24, 0, 0)
```

Using the `dateutil` module, you can parse dates from a variety of string formats.

[ ]:
```
from dateutil import parser
date = parser.parse("3rd of May, 2019")
date
```

[ ]:
```
datetime.datetime(2019, 5, 3, 0, 0)
```

Once we have the `datetime` object, we can do things such as printing the day of the week.

[ ]:
```
date.strftime('%A') # see the Python datetime documentation
```

[ ]:
```
'Friday'
```

Just as Python lists are suboptimal compared to NumPy arrays, lists of Python datetime objects are suboptimal compared to typed arrays of encoded dates.

Typed arrays of times: NumPy's datetime64

The `datetime64` dtype encodes the dates as 64-bit integers, and thus allows arrays of dates to be represented very compactly.

[ ]:
```
import numpy as np
date = np.array('2014-06-26', dtype=np.datetime64)
date
```

[ ]:
```
array('2014-06-26', dtype='datetime64[D]')
```

Once we have the date formatted like above, we can do vectorized operations on it.

[ ]:
```
date + np.arange(7)
```

[ ]:
```
array(['2014-06-26', '2014-06-27', '2014-06-28', '2014-06-29',
       '2014-06-30', '2014-07-01', '2014-07-02'], dtype='datetime64[D]')
```

Dates and times in Pandas: Best of both worlds

Pandas provides a `Timestamp` object, which combines the ease of use of `datetime` and `dateutil` with the efficient storage and vectorized interface of `numpy.datetime64`.

For a group of these `Timestamps` objects, Pandas can construct a `DateTimeIndex` that can be used to index data in a `Series` or `DataFrame`.

[ ]:
```
import pandas as pd
date = pd.to_datetime('5th of August, 1963')
date
```

[ ]:
```
Timestamp('1963-08-05 00:00:00')
```

[ ]:
```
date.strftime('%A')
```

[ ]:
```
'Monday'
```

We can perform Numpy-style vectorized operations directly on this object.

[ ]:
```
date + pd.to_timedelta(np.arange(12), 'D')
```

[ ]:
```
DatetimeIndex(['1963-08-05', '1963-08-06', '1963-08-07', '1963-08-08',
               '1963-08-09', '1963-08-10', '1963-08-11', '1963-08-12',
               '1963-08-13', '1963-08-14', '1963-08-15', '1963-08-16'],
              dtype='datetime64[ns]', freq=None)
```

Pandas Time Series: Indexing by Time

When the Pandas time series tools really become useful is when we *index data by timestamps*.

For example, we can construct a `Series` object that has time-indexed data.

[ ]:
```
import pandas as pd
index = pd.DatetimeIndex(['2015-4-25','2015-6-2','2016-7-19','2016-9-20'])

data = pd.Series([0, 1, 2, 3], index=index)
data
```

[ ]:
```
2015-04-25    0
2015-06-02    1
2016-07-19    2
2016-09-20    3
dtype: int64
```

Now that we have this data in `Series`, we can make use of any of the `Series` indexing patterns, passing values that can be coerced into dates.

[ ]:
```
data['2015-4-25':'2016-7-19']
```

[ ]:
```
2015-04-25    0
2015-06-02    1
2016-07-19    2
dtype: int64
```

We can also pass a year to get a slice of all data from that year.

[ ]:
```
data['2016']
```

[ ]:
```
2016-07-19    2
2016-09-20    3
dtype: int64
```

3.7.2 Pandas Time Series Data Structures

The fundamental Pandas data structures for working with time series data are as follows. * For *time stamps*, Pandas provides the `Timestamp` type. Please note that it is essentially a replacement for Python's native `datetime` object, however, it is based on the more efficient `numpy.datetime64` data type. The associated index structure is `DatetimeIndex`. * For *time periods*, Pandas provides the `Period` type. This encodes a fixed frequency interval based on `numpy.datetime64` object. The associated index structure is `PeriodIndex`. * For *time deltas* or *duration*, Pandas provides the `Timedelta` type. `Timedelta` is a more efficient replacement for Python's native `datetime.timedelta` type, and is based on `numpy.timedelta64`. The associated index structure is `TimedeltaIndex`.

The most fundamental of these date/time objects are the `Timestamp` and `DateTimeIndex` objects.

While these objects can be invoked directly, it is common to use the `pd.to_datetime()` function, which can parse a wide variety of formats.

Passing a single date to `pd.to_datetime()` gives us `Timestamp`;

Passing a series of dates by default gives us `DatetimeIndex`.

```
[ ]: from datetime import datetime
     dates = pd.to_datetime([datetime(2016,2,28), '15th of July, 2020',␣
      ↪'2018-Jun-5', '08-06-2017', '20140304'])
     dates
```

```
[ ]: DatetimeIndex(['2016-02-28', '2020-07-15', '2018-06-05', '2017-08-06',
                    '2014-03-04'],
                   dtype='datetime64[ns]', freq=None)
```

Any `DateTimeIndex` can be converted to a `PeriodIndex` with the `to_period()` function with the addition of frequency code.

In the following code, we will use 'D' to indicate daily frequency.

```
[ ]: dates.to_period('D')
```

```
[ ]: PeriodIndex(['2016-02-28', '2020-07-15', '2018-06-05', '2017-08-06',
                  '2014-03-04'],
                 dtype='period[D]', freq='D')
```

A `TimedeltaIndex` is created, for example, when one date is subtracted from another.

```
[ ]: dates - dates[0]
```

```
[ ]: TimedeltaIndex(['0 days', '1599 days', '828 days', '525 days', '-726 days'],
     dtype='timedelta64[ns]', freq=None)
```

Regular sequences: pd.date_range()

In order to make the creation of regular date sequences more convenient, Pandas provides a few functions as follows. * `pd.date_range()` for timestamps * `pd.period_range()` for periods * `pd.timedelta_range()` for time deltas.

Just like `range()` turns a start point, end point, and optional step size into a sequece, `pd.date_range()` accepts a start date, an end date, and an optional frequency code to create a regular sequence of dates.

By default, the frequency is one day.

```
[ ]: pd.date_range('2015-02-03', '2015-02-07')
```

```
[ ]: DatetimeIndex(['2015-02-03', '2015-02-04', '2015-02-05', '2015-02-06',
                    '2015-02-07'],
                   dtype='datetime64[ns]', freq='D')
```

The date range can be specified not with a start and endpoint, but with a startpoint and a number of periods.

```
[ ]: pd.date_range('2014-5-9', periods=7)
```

[ ]: DatetimeIndex(['2014-05-09', '2014-05-10', '2014-05-11', '2014-05-12',
               '2014-05-13', '2014-05-14', '2014-05-15'],
              dtype='datetime64[ns]', freq='D')

[ ]:

We can modify the spacing by altering the **freq** argument, which defaults to D.

For example, we can construct a range of hourly timestamps.

```
[ ]: pd.date_range('2015-9-25', periods=7, freq='H')
```

```
[ ]: DatetimeIndex(['2015-09-25 00:00:00', '2015-09-25 01:00:00',
               '2015-09-25 02:00:00', '2015-09-25 03:00:00',
               '2015-09-25 04:00:00', '2015-09-25 05:00:00',
               '2015-09-25 06:00:00'],
              dtype='datetime64[ns]', freq='H')
```

To create regular sequences of periods or time delta values, `pd.period_range()` and `pd.timedelta_range()` functions are useful.

```
[ ]: pd.period_range('2016-7', periods=7, freq='M')
```

```
[ ]: PeriodIndex(['2016-07', '2016-08', '2016-09', '2016-10', '2016-11', '2016-12',
             '2017-01'],
            dtype='period[M]', freq='M')
```

We can also create a sequence of durations increasing by an hour.

```
[ ]: pd.timedelta_range(0, periods=10, freq='H')
```

```
[ ]: TimedeltaIndex(['0 days 00:00:00', '0 days 01:00:00', '0 days 02:00:00',
                '0 days 03:00:00', '0 days 04:00:00', '0 days 05:00:00',
                '0 days 06:00:00', '0 days 07:00:00', '0 days 08:00:00',
                '0 days 09:00:00'],
               dtype='timedelta64[ns]', freq='H')
```

3.7.3 Frequencies and Offsets

The concept of frequency or date offset is fundamental to Pandas time series tools.

Just as we saw that D (Day) and H (Hour) codes, we can use such codes to specify any desired frequency spacing.

The table below shows the main codes available.

3.7.4 Resampling, Shifting, and Windowing

Pandas provides several additional time series-specific operations.

As Pandas was developed largely in financial context, it includes some very specific tools for financial data.

Let's take a look at an example. Here we load Microsoft's closing price history.

[3]:
```
import pandas as pd
msft = pd.read_csv('/content/drive/MyDrive/Python Training/Datasets/
↪MSFT_stock_for_Pandas_HW_Q8.csv')
msft.tail()
```

[3]:
```
            Date        Open        High  ...       Close   Adj Close    Volume
5404  2021-06-28  266.190002  268.899994  ...  268.720001  268.720001  19590000
5405  2021-06-29  268.869995  271.649994  ...  271.399994  271.399994  19937800
5406  2021-06-30  270.690002  271.359985  ...  270.899994  270.899994  21656500
5407  2021-07-01  269.609985  271.839996  ...  271.600006  271.600006  16725300
5408  2021-07-02  272.820007  278.000000  ...  277.649994  277.649994  26458000

[5 rows x 7 columns]
```

[7]:
```
msft.set_index('Date', inplace=True) # setting the Date column as explicit index
msft.head()
```

[7]:
```
                Open      High       Low     Close  Adj Close    Volume
Date
2000-01-04  56.78125  58.56250  56.12500  56.31250  35.684532  54119000
2000-01-05  55.56250  58.18750  54.68750  56.90625  36.060772  64059600
2000-01-06  56.09375  56.93750  54.18750  55.00000  34.852810  54976600
2000-01-07  54.31250  56.12500  53.65625  55.71875  35.308266  62013600
2000-01-10  56.71875  56.84375  55.68750  56.12500  35.565704  44963600
```

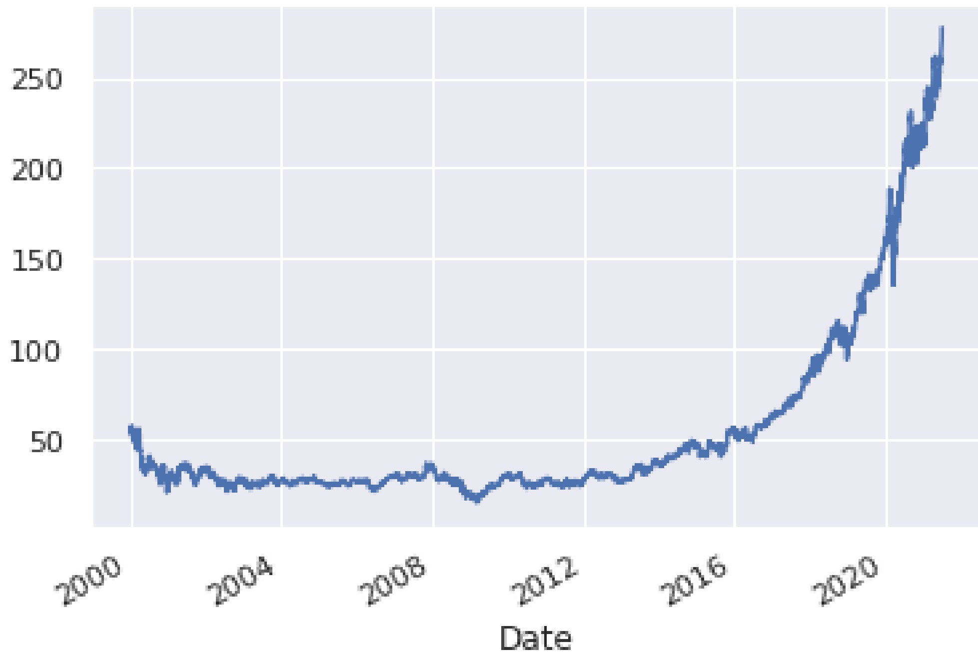
Let's use the closing price column data.

[14]:
```
msft_close = msft['Close']
msft_close.index = pd.to_datetime(msft_close.index) # converting the dtype of␣
↪index (Date) to DateTime type
print(msft_close.index.dtype)
```

```
datetime64[ns]
```

We will visualize this data using matplotlib.

[16]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()

msft_close.plot();
```



- **Resampling and converting frequencies**

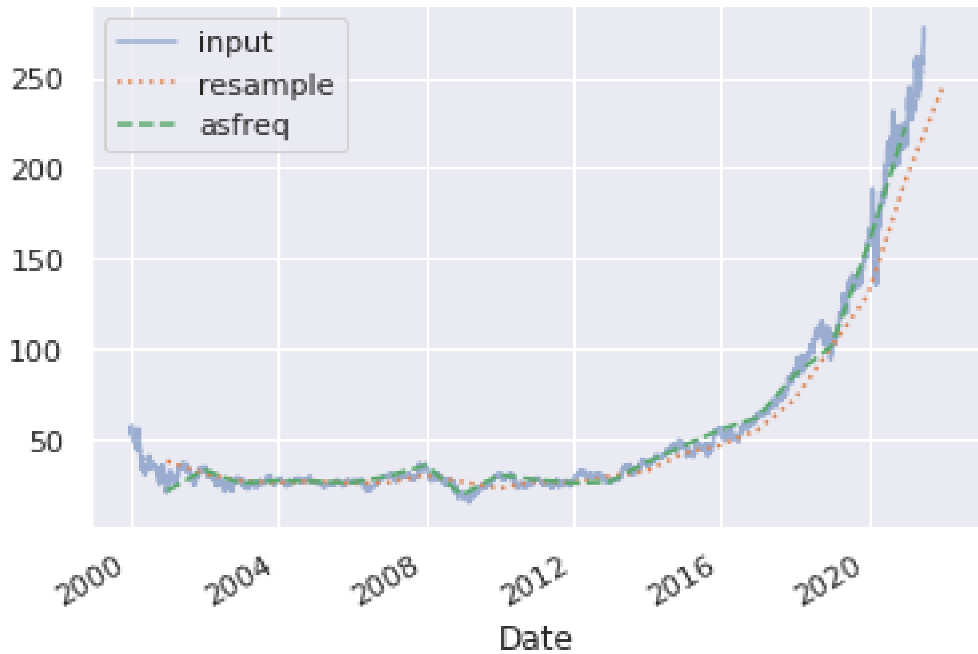
One common need for time series data is resampling at a higher or lower frequency.

We can do that using the `resample()` method, or the much simpler `asfreq()` method.

The main difference between the two is: * `resample()` is fundamentally a *data aggregation* * `asfreq()` is fundamentally a *data selection*.

[17]:
```
msft_close.plot(alpha=0.5, style='-')
msft_close.resample('BA').mean().plot(style=':')
msft_close.asfreq('BA').plot(style='--');
plt.legend(['input', 'resample', 'asfreq'],
           loc='upper left')
```

[17]:
```
<matplotlib.legend.Legend at 0x7f1cb6d44ed0>
```



[20]:
```
close_00_04 = msft_close['2000':'2004']
```

[27]:
```
from matplotlib.pyplot import figure
figure(figsize=(4, 3), dpi=100)
close_00_04.plot(alpha=0.5, label='closing price')
close_00_04.resample('BA').mean().plot(style='--', label='resample')
close_00_04.asfreq('BA').plot(style=':g', label='asfreq')
plt.legend();
```

Note the difference: at each point, `resample` reports the *average of the previous year*, while `asfreq` reports the *value at the end of the year*.

- **Time-shifts**

Another common time series-specific operation is shifting of data in time.

Pandas has the following method for computing this: * `shift()`: shifts the data

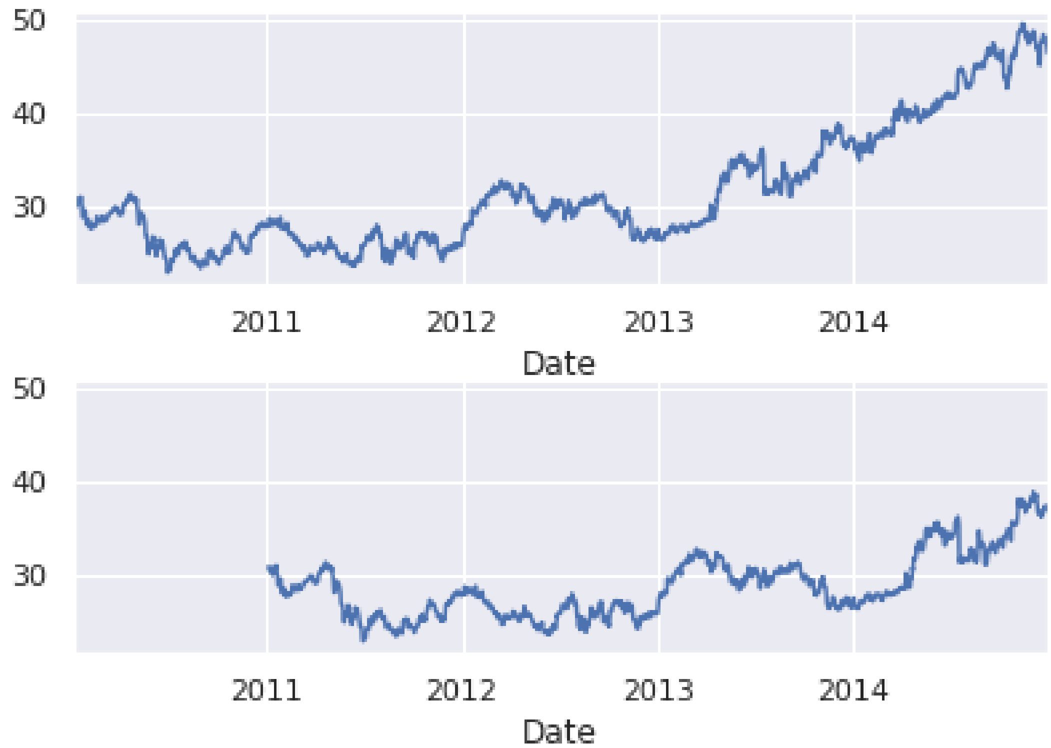
The shift is specified in multiples of frequency.

Let's select 5 years of `goog` data (2010 - 2014) and `shift()` by 365 days.

[28]:
```
msft10_14 = msft_close['2010':'2014']
msft10_14
```

[28]:
```
Date
2010-01-04    30.950001
2010-01-05    30.959999
2010-01-06    30.770000
2010-01-07    30.450001
2010-01-08    30.660000
                …
2014-12-24    48.139999
2014-12-26    47.880001
2014-12-29    47.450001
2014-12-30    47.020000
2014-12-31    46.450001
Name: Close, Length: 1258, dtype: float64
```

[29]:
```
import pandas as pd
import matplotlib.pyplot as plt

fig, ax = plt.subplots(2, sharey=True)
fig.tight_layout(pad=1.0)

# apply a frequency to the data
msft10_14 = msft10_14.asfreq('D', method='pad')

msft10_14.plot(ax=ax[0]);
msft10_14.shift(365).plot(ax=ax[1]); # data shifted by 365 days to the right
```



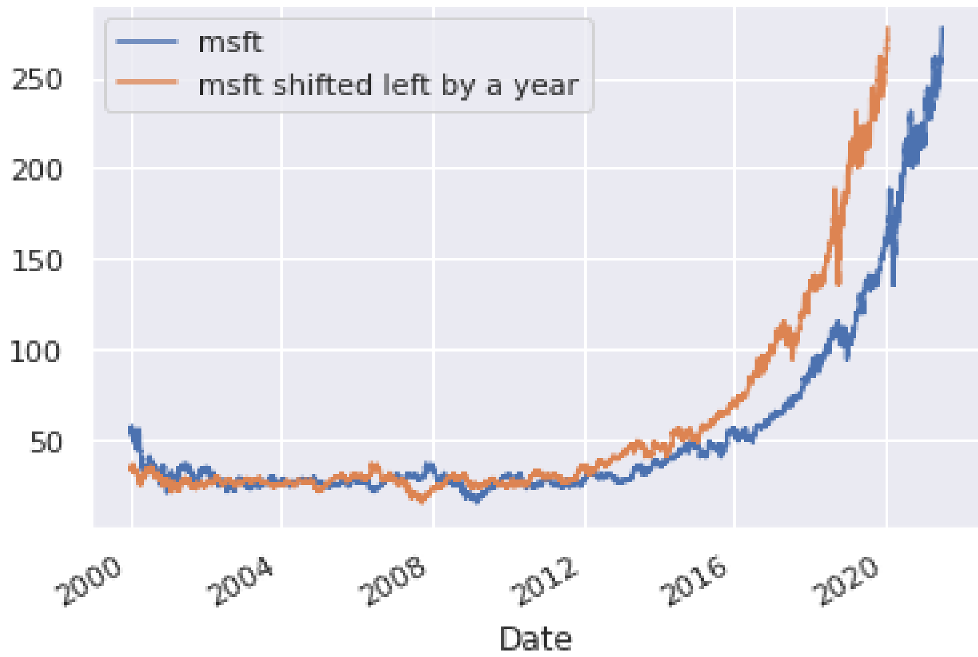
Here we see that **shift(365)** shifts the *data* by 365 days, pushing some of it off the end of the graph (and leaving NA values on the left end)

One common context for this type of shift is computing the differences over time.

For example, we can use shifted values to compute the one-year return on investment (ROI) for Microsoft stock over the course of the dataset.

[31]:
```
msft_close.plot()
msft_close.shift(-365).plot();plt.legend(['msft','msft shifted left by a year'])
```

[31]: <matplotlib.legend.Legend at 0x7f1cb6c57950>



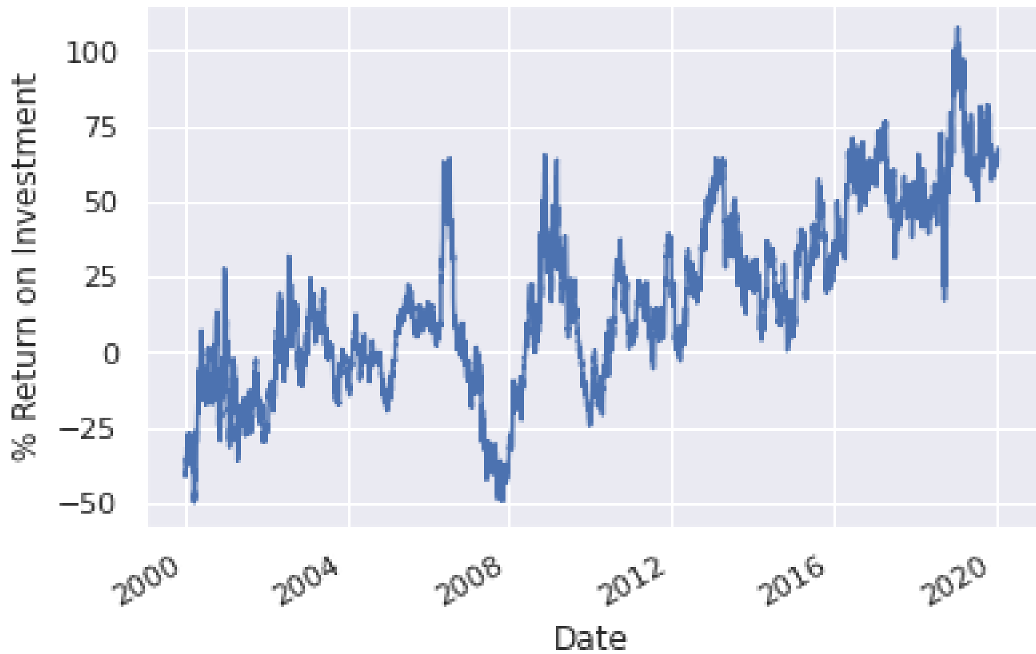
When we shift the graph by -365 days (orange curve), we are bringing the future back by one year if you look at a particular time point. So the blue curve represents the current price, and the orange curve represents the price one year into the future at a certain time point. Therfore, blue curve is the cost price (CP) and orange curve is the selling price (SP).

Profit/loss % (ROI) =

```
= (SP - CP)/CP * 100%
= (SP/CP - CP/CP) * 100%
= (SP/CP - 1) * 100%
```

In essence, wherever the orange curve is above the blue curve, there is profit; and there is loss in the converse case.

[34]:
```
ROI = ((msft_close.shift(-365) / msft_close) - 1) * 100

ROI.plot()
plt.ylabel('% Return on Investment');
```



This helps us see the overall trend in Microsoft stock.

3.7.5 Example: Visualizing Seattle Bicycle Counts

Now, let's look at another example of working with time series data.

Let's analyze the data on bicycle counts on Seattle's Fremont Bridge.

[ ]:
```
!curl -o FremontBridge.csv https://data.seattle.gov/api/views/65db-xm6k/rows.
↪csv?accessType=DOWNLOAD
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 4426k    0 4426k    0     0  1237k      0 --:--:--  0:00:03 --:--:-- 1237k
```

[ ]:
```
import pandas as pd

data = pd.read_csv('/content/FremontBridge.csv', index_col='Date',␣
↪parse_dates=True)
data.head()
```

[ ]:
```
                     Fremont Bridge Total  …  Fremont Bridge West Sidewalk
Date                                       …
2019-11-01 00:00:00                  12.0  …                           5.0
2019-11-01 01:00:00                   7.0  …                           7.0
2019-11-01 02:00:00                   1.0  …                           1.0
```

```
2019-11-01 03:00:00                    6.0  …                            0.0
2019-11-01 04:00:00                    6.0  …                            1.0

[5 rows x 3 columns]
```

For convenience, let's shorten the column names and add a 'Total' column.

[ ]:
```
data.columns
```

[ ]:
```
Index(['Fremont Bridge Total', 'Fremont Bridge East Sidewalk',
       'Fremont Bridge West Sidewalk'],
      dtype='object')
```

[ ]:
```
data.columns = ['Total', 'East', 'West']

data.head()
```

[ ]:
```
                     Total  East  West
Date
2019-11-01 00:00:00   12.0   7.0   5.0
2019-11-01 01:00:00    7.0   0.0   7.0
2019-11-01 02:00:00    1.0   0.0   1.0
2019-11-01 03:00:00    6.0   6.0   0.0
2019-11-01 04:00:00    6.0   5.0   1.0
```

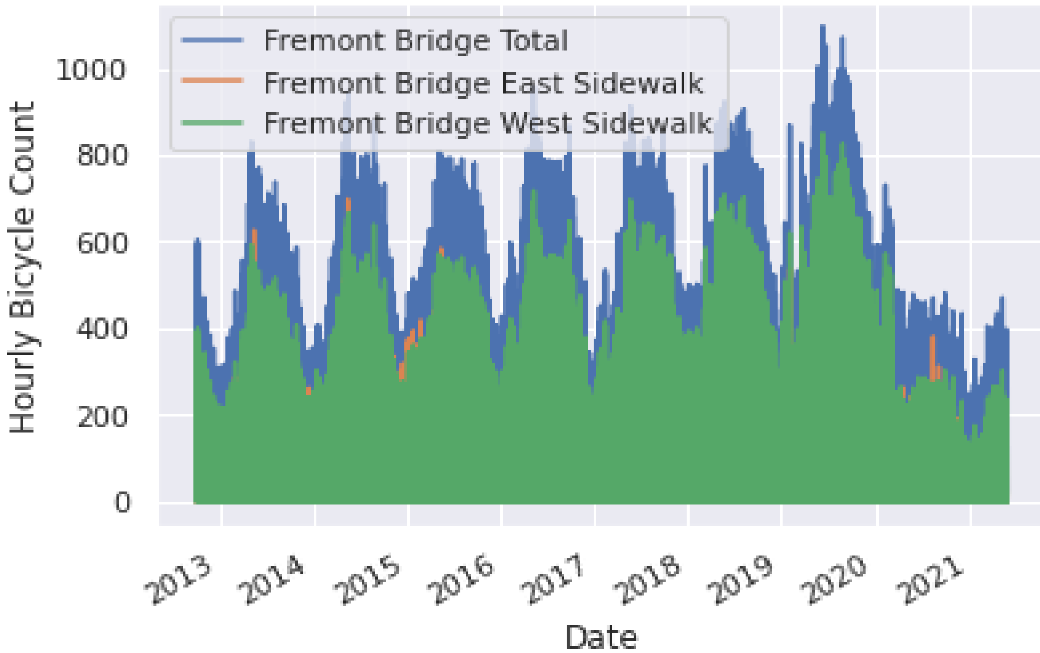
[ ]:
```
# Summary statistics
data.dropna().describe()
```

[ ]:
```
               Total          East           West
count  141400.000000  141400.00000  141400.000000
mean      111.169434      50.61628      60.553154
std       141.999671      65.46336      88.279627
min         0.000000       0.00000       0.000000
25%        14.000000       6.00000       7.000000
50%        60.000000      28.00000      30.000000
75%       145.000000      68.00000      74.000000
max      1097.000000     698.00000     850.000000
```

Visualizing the data

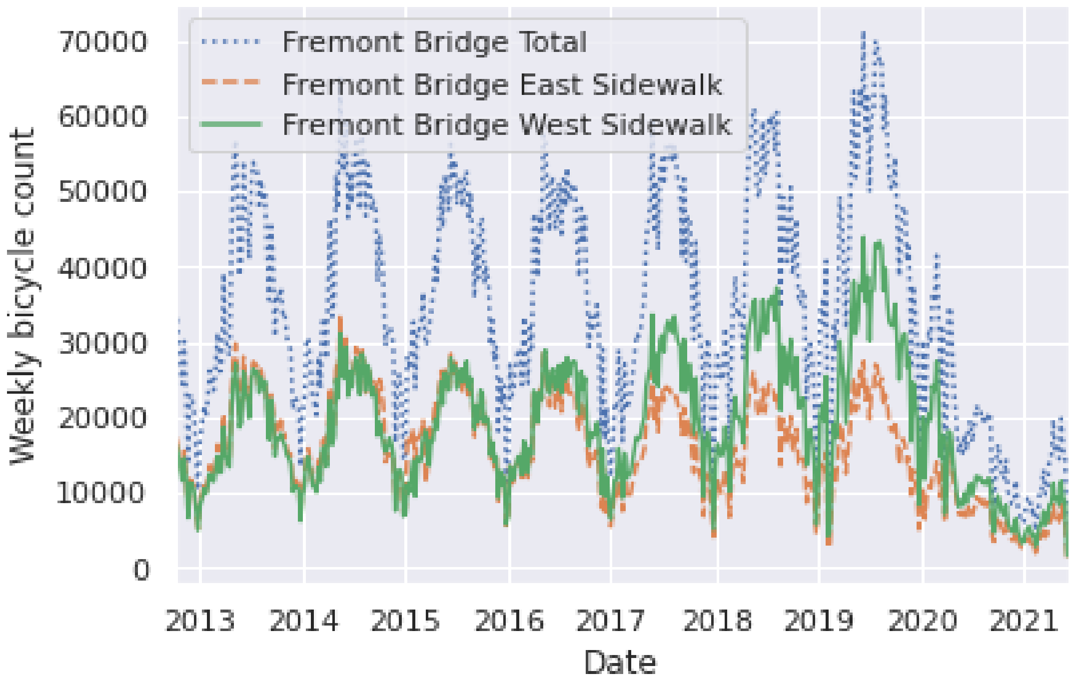
[ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt

import seaborn; seaborn.set()

data.plot()
plt.ylabel('Hourly Bicycle Count')
```

[ ]: Text(0, 0.5, 'Hourly Bicycle Count')



Let's resample the data by week.

[ ]:
```
weekly = data.resample('W').sum()
weekly.plot(style=[':', '--', '-'])
plt.ylabel('Weekly bicycle count');
```



[ ]:
```
weekly.shape
```

[ ]: (453, 3)

The above graph shows some interesting seasonal trends, such as, people bicycle more in the summer than in the winter. Also, the count is going down towards 2020/2021, possibly due to the pandemic.

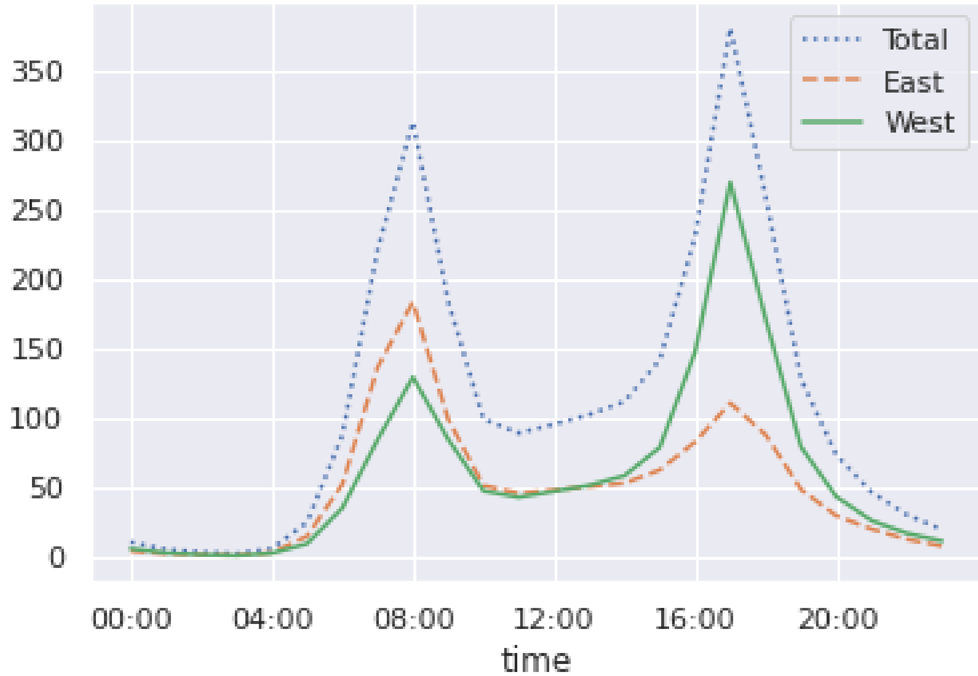
Digging into data

We can look at the average traffic as a function of the time of the day. We can do that by using the GroupBy functionality.

[ ]:
```
data.index.time[:30]
```

[ ]:
```
array([datetime.time(0, 0), datetime.time(1, 0), datetime.time(2, 0),
       datetime.time(3, 0), datetime.time(4, 0), datetime.time(5, 0),
       datetime.time(6, 0), datetime.time(7, 0), datetime.time(8, 0),
       datetime.time(9, 0), datetime.time(10, 0), datetime.time(11, 0),
       datetime.time(12, 0), datetime.time(13, 0), datetime.time(14, 0),
       datetime.time(15, 0), datetime.time(16, 0), datetime.time(17, 0),
       datetime.time(18, 0), datetime.time(19, 0), datetime.time(20, 0),
       datetime.time(21, 0), datetime.time(22, 0), datetime.time(23, 0),
       datetime.time(0, 0), datetime.time(1, 0), datetime.time(2, 0),
       datetime.time(3, 0), datetime.time(4, 0), datetime.time(5, 0)],
      dtype=object)
```

```
[ ]: import numpy as np

by_time = data.groupby(data.index.time).mean()
hourly_ticks = 4 * 60 * 60 * np.arange(6)
by_time.plot(xticks=hourly_ticks, style=[':', '--', '-']);
```



This shows a strongly bimodal distribution, with peaks around 8 AM and 5 PM.

```
[ ]: by_time # mean() of each hour
```

[ ]:

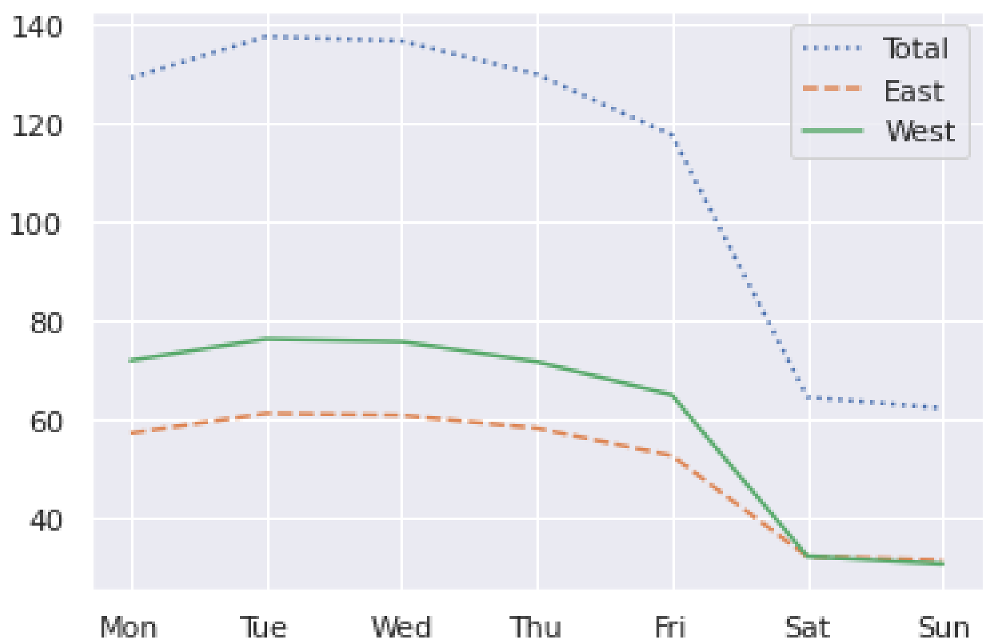
	Total	East	West
00:00:00	11.022569	4.661293	6.361276
01:00:00	5.826404	2.602410	3.223995
02:00:00	3.829563	1.835516	1.994047
03:00:00	2.826744	1.454777	1.371967
04:00:00	6.373664	3.360258	3.013406
05:00:00	25.116749	15.246055	9.870694
06:00:00	87.757679	52.421517	35.336162
07:00:00	220.851688	136.417275	84.434414
08:00:00	312.758188	183.376039	129.382148
09:00:00	183.970463	99.201833	84.768630
10:00:00	99.197250	51.524868	47.672382
11:00:00	89.156849	45.958072	43.198778
12:00:00	95.525547	48.208963	47.316585

```
13:00:00  102.429662   50.766672   51.662990
14:00:00  111.844535   53.215037   58.629498
15:00:00  141.609470   62.714358   78.895112
16:00:00  230.144094   82.829599  147.314494
17:00:00  380.210625  110.808045  269.402580
18:00:00  258.792430   87.882722  170.909708
19:00:00  127.981161   48.715547   79.265614
20:00:00   73.389172   29.844535   43.544637
21:00:00   46.755771   20.374915   26.380855
22:00:00   30.482179   13.093007   17.389172
23:00:00   20.019348    8.177868   11.841480
```

Let's see how things change based on the day of the week.

[ ]:
```
by_weekday = data.groupby(data.index.dayofweek).mean()
by_weekday.index = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
by_weekday.plot(style=[':', '--', '-'])
```

[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe916bfd350>
```



[ ]:
```
data.index.dayofweek.unique()
```

[ ]:
```
Int64Index([4, 5, 6, 0, 1, 2, 3], dtype='int64', name='Date')
```

[ ]:
```
by_weekday
```

[ ]:

	Total	East	West
Mon	129.231173	57.282006	71.949168
Tue	137.602626	61.269970	76.332656
Wed	136.706559	60.914798	75.791761
Thu	129.993575	58.286971	71.706603
Fri	117.685755	52.720105	64.965650
Sat	64.532154	32.246681	32.285474
Sun	62.275855	31.530342	30.745513

This shows a strong distinction between weekday and weekend totals, with around twice as many average riders on Monday through Friday than on the weekends.