What is Data Science?

Data science – discovery of data insight

Data science is a multidisciplinary blend of data inference, algorithm development, and technology in order to solve analytically complex problems.

This aspect of data science is all about uncovering findings from data. Diving in at a granular level to mine and understand complex behaviors, trends, and inferences.

It's about surfacing hidden insight that can help enable companies to make smarter business decisions.

Applications of Data Science

Recommender Systems

Netflix data mines movie viewing patterns to understand what drives user interest, and uses that to make decisions on which Netflix original series to produce.

Online Retailers such as Amazon, Flipkart identifies what are major customer segments within its base and the unique shopping behaviors within those segments, which helps to guide messaging to different market audiences.

Fraud and Risk Detection

Customer sentiment analysis

The analysts can perform the brand-customer sentiment analysis by data received from social networks and online services feedbacks.

Social media sources are readily available. That is why it is much easier to implement analytics on social platforms.

Sentiment analytics uses language processing to track words bearing a positive or negative attitude of a customer.

Sports to analyze player movements and team trends and make better coaching decisions.

Digital Advertisements (Targeted Advertising and retargeting)

The reason why digital ads have been able to get a lot higher CTR than traditional advertisements. They can be targeted based on user’s past behaviour.

This is the reason why you see ads of analytics trainings while your friend sees ad of apparels in the same place at the same time.

Image Recognition

Speech Recognition

Price Comparison Websites

At a basic level, these websites are being driven by lots and lots of data which is fetched using APIs and RSS Feeds.

Gaming

EA Sports, Zynga, Sony, Nintendo, Activision-Blizzard have led gaming experience to the next level using data science. Games are now designed using machine learning algorithms which improve / upgrade themselves as the player moves up to a higher level.

In motion gaming also, your opponent (computer) analyzes your previous moves and accordingly shapes up its game.

Airline Route Planning

Ride Sharing Services(such as Uber, Ola, Lyft) use Data Science

Self Driving Cars

Recommending new connections on Linkedin, Suggesting new people to follow on twitter.

Delivery logistics

Logistic companies like DHL, FedEx, UPS have used data science to improve their operational efficiency. Using data science, these companies have discovered the best routes to ship, the best suited time to deliver, the best mode of transport to choose thus leading to cost efficiency, and many more to mention.

Furthermore, the data that these companies generate using the GPS installed, provides them a lots of possibilities to explore using data science.

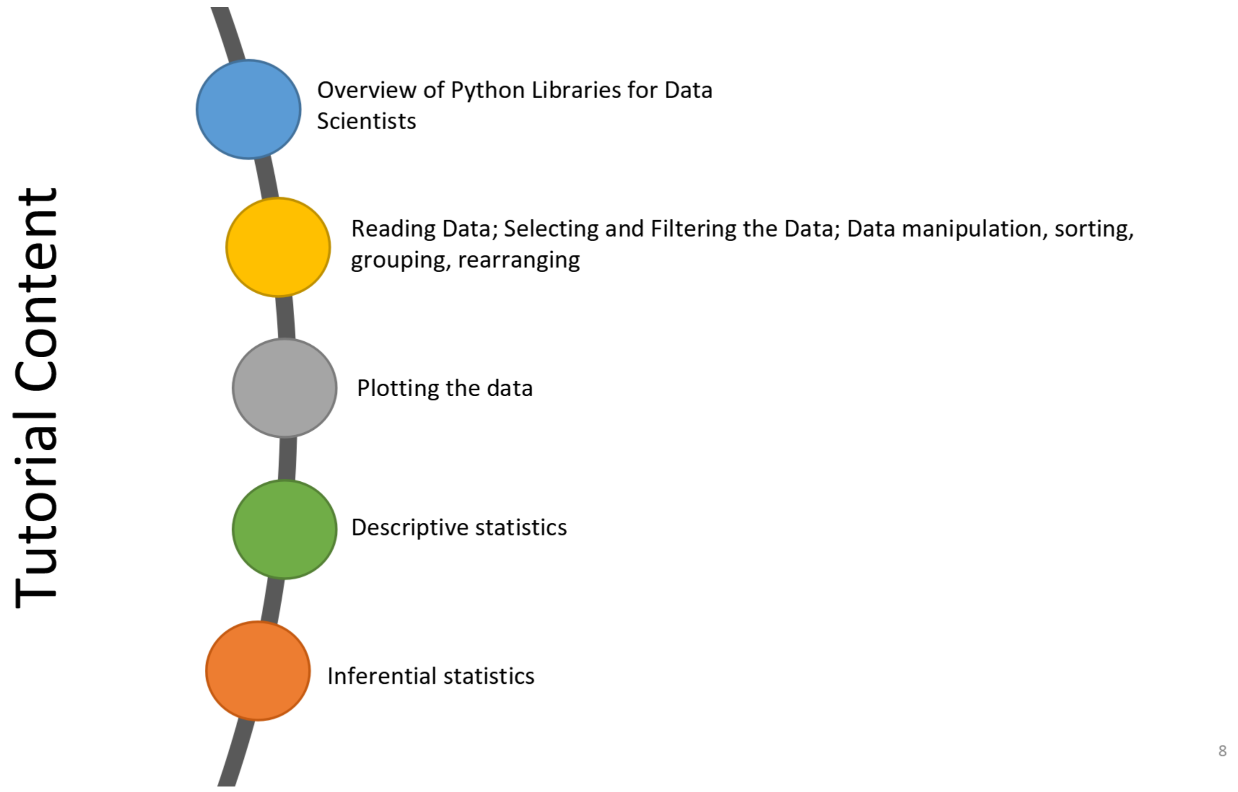
Dating websites (Ex. Tinder)

Bioinformatics

Urban Planning to solve traffic issues

Astrophysics

Public Health most likely emergency cases so as to provide better medical equipment and facility.



Tutorial Content

- Overview of Python Libraries for Data Scientists
- Reading Data; Selecting and Filtering the Data; Data manipulation, sorting, grouping, rearranging
- Plotting the data
- Descriptive statistics
- Inferential statistics

8

Python Libraries for Data Science

Many popular Python toolboxes/libraries:

NumPy SciPy Pandas SciKit-Learn

Visualization libraries:

matplotlib ggplot

```
                                                    and many more …
```

Python Libraries for Data Science

NumPy:

Numpy is a library that provides functions that are especially useful when you have to work with large arrays and matrices of numeric data, like doing matrix matrix multiplications. Also, Numpy is battle tested and optimized so that it runs fast, much faster than if you were working with Python lists directly.

provides vectorization of mathematical operations on arrays and matrices which significantly improves the performance

many other python libraries are built on NumPy

SciPy:

collection of algorithms for linear algebra, differential equations, numerical integration, optimization, statistics and more

part of SciPy Stack

built on NumPy

Pandas:

adds data structures and tools designed to work with table-like data (similar to Series and Data Frames in R)

provides tools for data manipulation: reshaping, merging, sorting, slicing, aggregation etc.

allows handling missing data

Handling data in a way that suits analysis

Scikit-Learn:

provides machine learning algorithms: classification, regression, clustering, model validation etc.

built on NumPy, SciPy and matplotlib

Matplotlib:

python 2D plotting library which produces publication quality figures in a variety of hard-copy formats

a set of functionalities similar to those of MATLAB

line plots, scatter plots, barcharts, histograms, pie charts etc.

relatively low-level; some effort needed to create advanced visualization

Let's Start with NumPy....

NumPy already comes installed with the Anaconda stack or you can install it using commands: pip install numpy (or) conda install numpy

In [1]:

```
import numpy as np #importing numpy and renaming as np for local use and ease
np.__version__     #check numpy version
```

Out[1]:

```
'1.14.0'
```

NumPy arrays:

NumPy arrays are different from Python lists. NumPy arrays can contain data of a single data type unlike lists in python. For eg. A python list can contain float as well as integer values at the same time

In [2]:

```
list = [0, 1.2 , 4, 3.43]
print(list)
```

```
[0, 1.2, 4, 3.43]
```

But for the same list if we create a numpy array, it would look like this:

In [3]:

```
print(np.array(list))
```

```
[0.   1.2  4.   3.43]
```

The data type of all the elements is upcasted to float.

You can explicitly set the data type of the resulting array by using the dtype keyword

In [4]:

```
np.array([1,2,3,4], dtype='float32')
```

Out[4]:

```
array([1., 2., 3., 4.], dtype=float32)
```

Creating NumPy arrays from scratch:

In [5]:

```
#Create a length-10 integer array filled with zeros
np.zeros(10,dtype=int)
```

Out[5]:

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

In [6]:

```
#Create a 3x5 floating point array filled with 1s
np.ones((3,5),dtype=float)
```

Out[6]:

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

In [7]:

```
#Create a 3x5 array filled with 3.14
np.full((3, 5), 3.14)
```

Out[7]:

```
array([[3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14]])
```

In [8]:

```
# Create an array filled with a linear sequence
# Starting at 0, ending at 20, stepping by 2
# (this is similar to the built-in range() function)
np.arange(0, 20, 2)
```

Out[8]:

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

In [9]:

```
# Create an array of five values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
```

Out[9]:

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

In [10]:

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

Out[10]:

```
array([[0.12212472, 0.82517557, 0.54541222],
       [0.90527964, 0.29078456, 0.17803106],
       [0.66933912, 0.20921547, 0.61450299]])
```

In [11]:

```
# Create a 3x3 array of random integers in the interval [0, 10)
np.random.randint(0, 10, (3, 3))
```

Out[11]:

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

In [12]:

```
# Create a 3x3 identity matrix
np.eye(3)
```

Out[12]:

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

Numpy Standard Data Types:

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2147483648 to 2147483647)
`int64`	Integer (-9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
`float16`	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa

Data type	Description
`float32`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128` .
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

NumPy array attributes:

In [13]:

```
import numpy as np
np.random.seed(0)  # seed for reproducibility

x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

In [14]:

```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
print("dtype:", x3.dtype)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
dtype: int32
```

Array indexing:

Accessing single elements:

If you are familiar with Python's standard list indexing, indexing in NumPy will feel quite familiar. In a one-dimensional array, the i^{th} value (counting from zero) can be accessed by specifying the desired index in square brackets, just as with Python lists:

In [15]:

```
print(x1)
print(x1[0])
print(x1[4])
print(x1[-1]) #prints the last element
print(x1[-2]) #2nd last element
```

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

In [16]:

```
print(x2)
print(x2[0, 0])
print(x2[2, 0])
print(x2[2, -1])
x2[0, 0] = 12
print(x2)
```

```
[[3 5 2 4]
 [7 6 8 8]
 [1 6 7 7]]
3
1
7
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

In [17]:

```
x1[0] = 3.14159  # this will be truncated! Guess why?
x1
```

Out[17]:

```
array([3, 0, 3, 3, 7, 9])
```

Array Slicing:

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the slice notation, marked by the colon (:) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array x, use this: x[start:stop:step] If any of these are unspecified, they default to the values start=0, stop=size of dimension, step=1. We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions.

In [18]:

```
x = np.arange(10)
x
```

Out[18]:

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

In [19]:

```
x[:5]  # first five elements
```

Out[19]:

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

In [20]:

```
x[4:7]  # middle sub-array
```

Out[20]:

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

In [21]:

```
x[::2]  # every other element
```

Out[21]:

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

In [22]:

```
x[1::2]  # every other element, starting at index 1
```

Out[22]:

```
array([1, 3, 5, 7, 9])
```

In [23]:

```
x[::-1]  # all elements, reversed
```

Out[23]:

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

In [24]:

```
x[5::-2]  # reversed every other from index 5
```

Out[24]:

```
array([5, 3, 1])
```

In [25]:

```
x2_sub = x2[:2, :2]
print(x2_sub)
```

```
[[12  5]
 [ 7  6]]
```

Reshaping Arrays:

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the reshape method. For example, if you want to put the numbers 1 through 9 in a 3×3 3×3 grid, you can do the following:

In [26]:

```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
```

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

In [27]:

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

# row vector via reshape
x.reshape((1, 3))
```

Out[27]:

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

In [28]:

```
x.reshape((3,1))
```

Out[28]:

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

Array Concatenation and Splitting

All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look at those operations here.

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate` , `np.vstack` , and `np.hstack` . `np.concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

In [29]:

```
x = np.array([1,2,3])
y = np.array([3,2,1])
np.concatenate([x,y])
```

Out[29]:

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

In [30]:

```
grid = np.array([[1,2,3],
                 [4,5,6]])
print(np.concatenate([grid,grid]))
print()
print(np.concatenate([grid,grid],axis=1))
```

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

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

Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split` , `np.hsplit` , and `np.vsplit` . For each of these, we can pass a list of indices giving the split points:

In [31]:

```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```

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

Notice that *N* split-points, leads to *N + 1* subarrays. The related functions `np.hsplit` and `np.vsplit` are similar:

In [32]:

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

Out[32]:

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

In [33]:

```
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

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

In [34]:

```
left, right = np.hsplit(grid, [2])
print(left)
print(right)
```

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

Exploring NumPy's UFuncs

Ufuncs exist in two flavors: *unary ufuncs*, which operate on a single input, and *binary ufuncs*, which operate on two inputs. We'll see examples of both these types of functions here.

Array arithmetic

NumPy's ufuncs feel very natural to use because they make use of Python's native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used:

In [35]:

```
x = np.arange(4)
print("x     =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
print("x / 2 =", x / 2)
print("x // 2 =", x // 2)  # floor division
```

```
x     = [0 1 2 3]
x + 5 = [5 6 7 8]
x - 5 = [-5 -4 -3 -2]
x * 2 = [0 2 4 6]
x / 2 = [0.  0.5 1.  1.5]
x // 2 = [0 0 1 1]
```

There is also a unary ufunc for negation, and a `**` operator for exponentiation, and a `%` operator for modulus:

In [36]:

```
print("-x     = ", -x)
print("x ** 2 = ", x ** 2)
print("x % 2  = ", x % 2)
```

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

In addition, these can be strung together however you wish, and the standard order of operations is respected:

In [37]:

```
-(0.5*x + 1) ** 2
```

Out[37]:

```
array([-1.  , -2.25, -4.  , -6.25])
```

Each of these arithmetic operations are simply convenient wrappers around specific functions built into NumPy; for example, the `+` operator is a wrapper for the `add` function:

In [38]:

```
np.add(x, 2)
```

Out[38]:

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

The following table lists the arithmetic operators implemented in NumPy:

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Absolute value

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

In [39]:

```
x = np.array([-2, -1, 0, 1, 2])
abs(x)
```

Out[39]:

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

In [40]:

```
np.abs(x) #You can also use np.absolute(x)
```

Out[40]:

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

This ufunc can also handle complex data, in which the absolute value returns the magnitude:

In [41]:

```
x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
np.abs(x)
```

Out[41]:

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

Trigonometric functions

NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions. We'll start by defining an array of angles:

In [42]:

```
theta = np.linspace(0, np.pi, 3) #Divides the range of 0 to pi into 2 equal parts giving 3
```

Now we can compute some trigonometric functions on these values:

In [43]:

```
print("theta      = ", theta)
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("tan(theta) = ", np.tan(theta))
```

```
theta      =  [0.         1.57079633 3.14159265]
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

Another common type of operation available in a NumPy ufunc are the exponentials:

In [44]:

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

The inverse of the exponentials, the logarithms, are also available. The basic `np.log` gives the natural logarithm; if you prefer to compute the base-2 logarithm or the base-10 logarithm, these are available as well:

In [45]:

```
x = [1, 2, 4, 10]
print("x        =", x)
print("ln(x)    =", np.log(x))
print("log2(x)  =", np.log2(x))
print("log10(x) =", np.log10(x))
```

```
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.        ]
```

Aggregates

For binary ufuncs, there are some interesting aggregates that can be computed directly from the object. For example, if we'd like to *reduce* an array with a particular operation, we can use the `reduce` method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result remains.

For example, calling `reduce` on the `add` ufunc returns the sum of all elements in the array:

In [46]:

```
x = np.arange(1, 6)
np.add.reduce(x)
```

Out[46]:

```
15
```

In [47]:

```
np.multiply.reduce(x)
```

Out[47]:

```
120
```

If we'd like to store all the intermediate results of the computation, we can instead use `accumulate` :

In [48]:

```
np.add.accumulate(x)
```

Out[48]:

```
array([ 1,  3,  6, 10, 15], dtype=int32)
```

In [49]:

```
np.multiply.accumulate(x)
```

Out[49]:

```
array([  1,   2,   6,  24, 120], dtype=int32)
```

Aggregations: Min, Max, and Everything In Between

Often when faced with a large amount of data, a first step is to compute summary statistics for the data in question. Perhaps the most common summary statistics are the mean and standard deviation, which allow you to summarize the "typical" values in a dataset, but other aggregates are useful as well (the sum, product, median, minimum and maximum, quantiles, etc.).

NumPy has fast built-in aggregation functions for working on arrays; we'll discuss and demonstrate some of them here.

Summing the Values in an Array

As a quick example, consider computing the sum of all values in an array. Python itself can do this using the built-in `sum` function wheras `numpy` has its own `np.sum()` function. See the difference in time.

In [50]:

```
big_array = np.random.rand(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
```

```
274 ms ± 19.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.93 ms ± 112 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Minimum and Maximum

Similarly, Python has built-in `min` and `max` functions, used to find the minimum value and maximum value of any given array:

In [51]:

```
%timeit min(big_array)
%timeit np.min(big_array)
```

```
114 ms ± 1.93 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
772 µs ± 26.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Multi dimensional aggregates

One common type of aggregation operation is an aggregate along a row or column. Say you have some data stored in a two-dimensional array:

In [52]:

```
M = np.random.random((3, 4))
print(M)
M.sum()
```

```
[[0.0049466  0.25863997 0.62346477 0.90474173]
 [0.71661557 0.699582   0.80401456 0.60471376]
 [0.43905815 0.73525983 0.3703232  0.57361603]]
```

Out[52]:

```
6.734976167538792
```

Aggregation functions take an additional argument specifying the *axis* along which the aggregate is computed. For example, we can find the minimum value within each column by specifying `axis=0` :

In [53]:

```
M.min(axis=0)
```

Out[53]:

```
array([0.0049466 , 0.25863997, 0.3703232 , 0.57361603])
```

In [54]:

```
M.max(axis=1)
```

Out[54]:

```
array([0.90474173, 0.80401456, 0.73525983])
```

The way the axis is specified here can be confusing to users coming from other languages. The `axis` keyword specifies the *dimension of the array that will be collapsed*, rather than the dimension that will be returned. So specifying `axis=0` means that the first axis will be collapsed: for two-dimensional arrays, this means that values within each column will be aggregated.

Other aggregation functions

NumPy provides many other aggregation functions, but we won't discuss them in detail here. Additionally, most aggregates have a `NaN` -safe counterpart that computes the result while ignoring missing values, which are marked by the special IEEE floating-point `NaN` value. Some of these `NaN` -safe functions were not added until NumPy 1.8, so they will not be available in older NumPy versions.

The following table provides a list of useful aggregation functions available in NumPy:

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value

Function Name	NaN-safe Version	Description
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

We will see these aggregates often throughout the rest of the book.

Computation on Arrays: Broadcasting

We saw in the previous section how NumPy's universal functions can be used to *vectorize* operations and thereby remove slow Python loops. Another means of vectorizing operations is to use NumPy's *broadcasting* functionality. Broadcasting is simply a set of rules for applying binary ufuncs (e.g., addition, subtraction, multiplication, etc.) on arrays of different sizes.

Introducing Broadcasting

Recall that for arrays of the same size, binary operations are performed on an element-by-element basis:

In [55]:

```
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

Out[55]:

```
array([5, 6, 7])
```

Broadcasting allows these types of binary operations to be performed on arrays of different sizes–for example, we can just as easily add a scalar (think of it as a zero-dimensional array) to an array:

In [56]:

```
a + 5
```

Out[56]:

```
array([5, 6, 7])
```

We can think of this as an operation that stretches or duplicates the value `5` into the array `[5, 5, 5]`, and adds the results. The advantage of NumPy's broadcasting is that this duplication of values does not actually take place, but it is a useful mental model as we think about broadcasting.

We can similarly extend this to arrays of higher dimension. Observe the result when we add a one-dimensional array to a two-dimensional array:

In [57]:

```
M = np.ones((3, 3))
M + a
```

Out[57]:

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

Here the one-dimensional array `a` is stretched, or broadcast across the second dimension in order to match the shape of `M`.

While these examples are relatively easy to understand, more complicated cases can involve broadcasting of both arrays. Consider the following example:

In [58]:

```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]
```

In [59]:

```
a + b
```

Out[59]:

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

Just as before we stretched or broadcasted one value to match the shape of the other, here we've stretched *both* `a` and `b` to match a common shape, and the result is a two-dimensional array! The geometry of these examples is visualized in the following figure

Broadcasting Visual

Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

- Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.
- Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
- Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

To make these rules clear, let's consider a few examples in detail.

Broadcasting example 1

Let's look at adding a two-dimensional array to a one-dimensional array:

In [60]:

```
M = np.ones((2, 3))
a = np.arange(3)
```

Let's consider an operation on these two arrays. The shape of the arrays are

- `M.shape = (2, 3)`
- `a.shape = (3,)`

We see by rule 1 that the array `a` has fewer dimensions, so we pad it on the left with ones:

- `M.shape -> (2, 3)`
- `a.shape -> (1, 3)`

By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

- `M.shape -> (2, 3)`
- `a.shape -> (2, 3)`

The shapes match, and we see that the final shape will be `(2, 3)`:

In [61]:

```
M + a
```

Out[61]:

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

Broadcasting example 2

Let's take a look at an example where both arrays need to be broadcast:

In [62]:

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

Again, we'll start by writing out the shape of the arrays:

- `a.shape = (3, 1)`
- `b.shape = (3,)`

Rule 1 says we must pad the shape of `b` with ones:

- `a.shape -> (3, 1)`
- `b.shape -> (1, 3)`

And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

- `a.shape -> (3, 3)`
- `b.shape -> (3, 3)`

Because the result matches, these shapes are compatible. We can see this here:

In [63]:
```
a + b
```
Out[63]:
```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Boolean operations on NumPy array:

In [64]:
```
x = np.array([1, 2, 3, 4, 5])
```
In [65]:
```
x < 3  # less than
```
Out[65]:
```
array([ True,  True, False, False, False])
```
In [66]:
```
x >= 3  # greater than equal to
```
Out[66]:
```
array([False, False,  True,  True,  True])
```

As in the case of arithmetic operators, the comparison operators are implemented as ufuncs in NumPy; for example, when you write `x < 3` , internally NumPy uses `np.less(x, 3)` . A summary of the comparison operators and their equivalent ufunc is shown here:

| Operator | Equivalent ufunc || Operator | Equivalent ufunc | |---------------|---------------------||---------------|---------------------| | `==` | `np.equal` || `!=` | `np.not_equal` || `<` | `np.less` || `<=` | `np.less_equal` || `>` | `np.greater` || `>=` | `np.greater_equal` |

In [67]:
```
# how many values less than 6?
np.count_nonzero(x < 6)
```
Out[67]:
```
5
```

In [68]:

```
np.sum(x < 6)
```

Out[68]:

```
5
```

In [69]:

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

Out[69]:

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

In [70]:

```
# how many values less than 6 in each row?
np.sum(x < 6, axis=1)
```

Out[70]:

```
array([4, 2, 2])
```

In [71]:

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

Out[71]:

```
True
```

In [72]:

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

Out[72]:

```
True
```

Sorting NumPy Arrays:

Although Python has built-in sort and sorted functions to work with lists, we won't discuss them here because NumPy's np.sort function turns out to be much more efficient and useful for our purposes. By default np.sort uses an [NlogN] O[NlogN] , quicksort algorithm, though mergesort and heapsort are also available. For most applications, the default quicksort is more than sufficient. To return a sorted version of the array without modifying the input, you can use np.sort:

In [73]:

```
x = np.array([2, 1, 4, 3, 5])
np.sort(x) #doesn't sort inplace
```

Out[73]:

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

In [74]:

```
x.sort() #sorts inplace
print(x)
```

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

And with that we stop our discussion on NumPy. There are many other functions not mentioned above in the numpy library. We encourage you to look up the documentation for the same.

You can find the documentation here : https://docs.scipy.org/doc/numpy/ (https://docs.scipy.org/doc/numpy/)

Pandas:

Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a DataFrame. DataFrames are essentially multidimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data. As well as offering a convenient storage interface for labeled data, Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.

You can install pandas by:

1. Open cmd
2. Type 'pip install pandas' without quotes and click enter

Once Pandas is installed, you can import it and check the version:

In [75]:

```
import pandas as pd #importing pandas and renaming it locally for ease
pd.__version__
```

Out[75]:

```
'0.22.0'
```

The Pandas Series Object

A Pandas `Series` is a one-dimensional array of indexed data. It can be created from a list or array as follows:

In [76]:

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

Out[76]:

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

As we see in the output, the `Series` wraps both a sequence of values and a sequence of indices, which we can access with the `values` and `index` attributes. The `values` are simply a familiar NumPy array:

In [77]:

```
data.values
```

Out[77]:

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

The `index` is an array-like object of type `pd.Index` , which we'll discuss in more detail momentarily.

In [78]:

```
data.index
```

Out[78]:

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

Like with a NumPy array, data can be accessed by the associated index via the familiar Python square-bracket notation:

In [79]:

```
data[1]
```

Out[79]:

```
0.5
```

In [80]:

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

Out[80]:

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

From what we've seen so far, it may look like the `Series` object is basically interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the Numpy Array has an *implicitly defined* integer index used to access the values, the Pandas `Series` has an *explicitly defined* index associated with the values.

This explicit index definition gives the `Series` object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:

In [81]:

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

Out[81]:

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

In [82]:

```
data['b']
```

Out[82]:

```
0.5
```

You can think of a Pandas Series a bit like a specialization of a Python dictionary. A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a Series is a structure which maps typed keys to a set of typed values.

In [83]:

```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)
population
```

Out[83]:

```
California    38332521
Florida       19552860
Illinois      12882135
New York      19651127
Texas         26448193
dtype: int64
```

In [84]:

```
population['California']
```

Out[84]:

```
38332521
```

Constructing Series objects

We've already seen a few ways of constructing a Pandas `Series` from scratch; all of them are some version of the following:

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

where `index` is an optional argument, and `data` can be one of many entities.

For example, `data` can be a list or NumPy array, in which case `index` defaults to an integer sequence:

In [85]:

```
pd.Series([2, 4, 6])
```

Out[85]:

```
0    2
1    4
2    6
dtype: int64
```

In [86]:

```
pd.Series(5, index=[100, 200, 300])
```

Out[86]:

```
100    5
200    5
300    5
dtype: int64
```

The Pandas DataFrame Object

The next fundamental structure in Pandas is the `DataFrame` . Like the `Series` object discussed in the previous section, the `DataFrame` can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary. We'll now take a look at each of these perspectives.

DataFrame as a generalized NumPy array

If a `Series` is an analog of a one-dimensional array with flexible indices, a `DataFrame` is an analog of a two-dimensional array with both flexible row indices and flexible column names. Just as you might think of a two-dimensional array as an ordered sequence of aligned one-dimensional columns, you can think of a `DataFrame` as a sequence of aligned `Series` objects. Here, by "aligned" we mean that they share the same index.

To demonstrate this, let's first construct a new `Series` listing the area of each of the five states discussed in the previous section:

In [87]:

```
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
area
```

Out[87]:

```
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
dtype: int64
```

Now that we have this along with the `population` Series from before, we can use a dictionary to construct a single two-dimensional object containing this information:

In [88]:

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

Out[88]:

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

In [89]:

```
print("Indices : "+str(states.index))
print("Columns : "+str(states.columns))
```

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

DataFrame as specialized dictionary

Similarly, we can also think of a `DataFrame` as a specialization of a dictionary. Where a dictionary maps a key to a value, a `DataFrame` maps a column name to a `Series` of column data. For example, asking for the `'area'` attribute returns the `Series` object containing the areas we saw earlier:

In [90]:

```
states['area']
```

Out[90]:

```
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

Notice the potential point of confusion here: in a two-dimesnional NumPy array, `data[0]` will return the first *row*. For a `DataFrame` , `data['col0']` will return the first *column*. Because of this, it is probably better to think about `DataFrame` s as generalized dictionaries rather than generalized arrays, though both ways of looking at the situation can be useful.

Constructing DataFrame objects

A Pandas `DataFrame` can be constructed in a variety of ways. Here we'll give several examples.

From a single Series object

A `DataFrame` is a collection of `Series` objects, and a single-column `DataFrame` can be constructed from a single `Series` :

In [91]:

```
pd.DataFrame(population, columns=['population'])
```

Out[91]:

	population
California	38332521
Florida	19552860
Illinois	12882135
New York	19651127
Texas	26448193

From a list of dicts

Any list of dictionaries can be made into a `DataFrame` . We'll use a simple list comprehension to create some data:

In [92]:

```
data = [{'a': i, 'b': 2 * i}
        for i in range(3)]
pd.DataFrame(data)
```

Out[92]:

	a	b
0	0	0
1	1	2
2	2	4

Even if some keys in the dictionary are missing, Pandas will fill them in with `NaN` (i.e., "not a number") values:

In [93]:

```
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

Out[93]:

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

From a dictionary of Series objects

As we saw before, a `DataFrame` can be constructed from a dictionary of `Series` objects as well:

In [94]:

```
pd.DataFrame(np.random.rand(3, 2),
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])
```

Out[94]:

	foo	bar
a	0.784029	0.000750
b	0.508583	0.444828
c	0.021725	0.465671

From a NumPy structured array

A Pandas `DataFrame` operates much like a structured array, and can be created directly from one:

In [95]:

```
A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
pd.DataFrame(A)
```

Out[95]:

	A	B
0	0	0.0
1	0	0.0
2	0	0.0

Series as dictionary

Like a dictionary, the `Series` object provides a mapping from a collection of keys to a collection of values:

In [96]:

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

Out[96]:

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

In [97]:

```
data['b']
```

Out[97]:

```
0.5
```

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

In [98]:

```
'a' in data
```

Out[98]:

```
True
```

In [99]:

```
data.keys()
```

Out[99]:

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

In [100]:

```
data['e'] = 1.25
data
```

Out[100]:

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

This easy mutability of the objects is a convenient feature: under the hood, Pandas is making decisions about memory layout and data copying that might need to take place; the user generally does not need to worry about these issues.

Series as one-dimensional array

A `Series` builds on this dictionary-like interface and provides array-style item selection via the same basic mechanisms as NumPy arrays – that is, *slices*, *masking*, and *fancy indexing*. Examples of these are as follows:

In [101]:

```
# slicing by explicit index
data['a':'c']
```

Out[101]:

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

In [102]:

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

Out[102]:

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

Indexers: loc, iloc, and ix

These slicing and indexing conventions can be a source of confusion. For example, if your `Series` has an explicit integer index, an indexing operation such as `data[1]` will use the explicit indices, while a slicing operation like `data[1:3]` will use the implicit Python-style index.

In [103]:

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
```

Out[103]:

```
1    a
3    b
5    c
dtype: object
```

In [104]:

```
# explicit index when indexing
data[1]
```

Out[104]:

```
'a'
```

In [105]:

```
# implicit index when slicing
data[1:3]
```

Out[105]:

```
3    b
5    c
dtype: object
```

Because of this potential confusion in the case of integer indexes, Pandas provides some special *indexer* attributes that explicitly expose certain indexing schemes. These are not functional methods, but attributes that expose a particular slicing interface to the data in the `Series` .

First, the `loc` attribute allows indexing and slicing that always references the explicit index:

In [106]:

```
data.loc[1]
```

Out[106]:

```
'a'
```

In [107]:

```
data.loc[1:3]
```

Out[107]:

```
1    a
3    b
dtype: object
```

The iloc attribute allows indexing and slicing that always references the implicit Python-style index:

In [108]:

```
data.iloc[1]
```

Out[108]:

```
'b'
```

In [109]:

```
data.iloc[1:3]
```

Out[109]:

```
3    b
5    c
dtype: object
```

A third indexing attribute, `ix`, is a hybrid of the two, and for `Series` objects is equivalent to standard `[]`-based indexing. The purpose of the `ix` indexer will become more apparent in the context of `DataFrame` objects, which we will discuss in a moment.

One guiding principle of Python code is that "explicit is better than implicit." The explicit nature of `loc` and `iloc` make them very useful in maintaining clean and readable code; especially in the case of integer indexes, I recommend using these both to make code easier to read and understand, and to prevent subtle bugs due to the mixed indexing/slicing convention.

Data Selection in DataFrame

Recall that a `DataFrame` acts in many ways like a two-dimensional or structured array, and in other ways like a dictionary of `Series` structures sharing the same index. These analogies can be helpful to keep in mind as we explore data selection within this structure.

DataFrame as a dictionary

The first analogy we will consider is the `DataFrame` as a dictionary of related `Series` objects. Let's return to our example of areas and populations of states:

In [110]:

```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

Out[110]:

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

The individual Series that make up the columns of the DataFrame can be accessed via dictionary-style indexing of the column name:

In [111]:

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

Out[111]:

```
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

In [112]:

```
data.area
```

Out[112]:

```
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

In [113]:

```
#This attribute-style column access actually accesses the exact same object as the dictiona
data.area is data['area']
```

Out[113]:

```
True
```

Though this is a useful shorthand, keep in mind that it does not work for all cases! For example, if the column names are not strings, or if the column names conflict with methods of the `DataFrame`, this attribute-style access is not possible. For example, the `DataFrame` has a `pop()` method, so `data.pop` will point to this rather than the `"pop"` column:

In [114]:

```
data.pop is data['pop']
```

Out[114]:

```
False
```

In [115]:

```
data['density'] = data['pop'] / data['area']
data
```

Out[115]:

	area	pop	density
California	423967	38332521	90.413926
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

DataFrame as two-dimensional array

As mentioned previously, we can also view the `DataFrame` as an enhanced two-dimensional array. We can examine the raw underlying data array using the `values` attribute:

In [116]:

```
data.values
```

Out[116]:

```
array([[4.23967000e+05, 3.83325210e+07, 9.04139261e+01],
       [1.70312000e+05, 1.95528600e+07, 1.14806121e+02],
       [1.49995000e+05, 1.28821350e+07, 8.58837628e+01],
       [1.41297000e+05, 1.96511270e+07, 1.39076746e+02],
       [6.95662000e+05, 2.64481930e+07, 3.80187404e+01]])
```

In [117]:

```
data.T
```

Out[117]:

	California	Florida	Illinois	New York	Texas
area	4.239670e+05	1.703120e+05	1.499950e+05	1.412970e+05	6.956620e+05
pop	3.833252e+07	1.955286e+07	1.288214e+07	1.965113e+07	2.644819e+07
density	9.041393e+01	1.148061e+02	8.588376e+01	1.390767e+02	3.801874e+01

When it comes to indexing of `DataFrame` objects, however, it is clear that the dictionary-style indexing of columns precludes our ability to simply treat it as a NumPy array. In particular, passing a single index to an array accesses a row:

In [118]:

```
data.values[0]
```

Out[118]:

```
array([4.23967000e+05, 3.83325210e+07, 9.04139261e+01])
```

In [119]:

```
# passing a single "index" to a DataFrame accesses a column
data['area']
```

Out[119]:

```
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

Thus for array-style indexing, we need another convention. Here Pandas again uses the `loc` ,and `iloc` indexers mentioned earlier. Using the `iloc` indexer, we can index the underlying array as if it is a simple NumPy array (using the implicit Python-style index), but the `DataFrame` index and column labels are maintained in the result:

In [120]:

```
data.iloc[:3, :2]
```

Out[120]:

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

In [121]:

```
data.loc[:'Illinois', :'pop']
```

Out[121]:

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

Any of these indexing conventions may also be used to set or modify values; this is done in the standard way that you might be accustomed to from working with NumPy:

In [122]:

```
data.iloc[0, 2] = 90
data
```

Out[122]:

	area	pop	density
California	423967	38332521	90.000000
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

Index alignment in DataFrame

A similar type of alignment takes place for *both* columns and indices when performing operations on `DataFrame` s:

In [123]:

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

Out[123]:

	A	B
0	12	1
1	6	7

In [124]:

```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                 columns=['A','B','C'])
B
```

Out[124]:

	A	B	C
0	7	8	1
1	5	9	8
2	9	4	3

In [125]:

```
A+B
```

Out[125]:

	A	B	C
0	19.0	9.0	NaN
1	11.0	16.0	NaN
2	NaN	NaN	NaN

Notice that indices are aligned correctly irrespective of their order in the two objects, and indices in the result are sorted. As was the case with `Series`, we can use the associated object's arithmetic method and pass any desired `fill_value` to be used in place of missing entries. Here we'll fill with the mean of all values in `A` (computed by first stacking the rows of `A`):

In [126]:

```
fill = A.stack().mean()
A.add(B, fill_value=fill)
```

Out[126]:

	A	B	C
0	19.0	9.0	7.5
1	11.0	16.0	14.5
2	15.5	10.5	9.5

The following table lists Python operators and their equivalent Pandas object methods:

Python Operator	Pandas Method(s)
`+`	`add()`
`-`	`sub()`, `subtract()`
`*`	`mul()`, `multiply()`
`/`	`truediv()`, `div()`, `divide()`

Python Operator	Pandas Method(s)
`//`	`floordiv()`
`%`	`mod()`
`**`	`pow()`

Handling Missing Data

The difference between data found in many tutorials and data in the real world is that real-world data is rarely clean and homogeneous. In particular, many interesting datasets will have some amount of data missing. To make matters even more complicated, different data sources may indicate missing data in different ways.

In this section, we will discuss some general considerations for missing data, discuss how Pandas chooses to represent it, and demonstrate some built-in Pandas tools for handling missing data in Python. Here and throughout the book, we'll refer to missing data in general as *null*, *NaN*, or *NA* values.

Trade-Offs in Missing Data Conventions

There are a number of schemes that have been developed to indicate the presence of missing data in a table or DataFrame. Generally, they revolve around one of two strategies: using a *mask* that globally indicates missing values, or choosing a *sentinel value* that indicates a missing entry.

In the masking approach, the mask might be an entirely separate Boolean array, or it may involve appropriation of one bit in the data representation to locally indicate the null status of a value.

In the sentinel approach, the sentinel value could be some data-specific convention, such as indicating a missing integer value with -9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating-point value with NaN (Not a Number), a special value which is part of the IEEE floating-point specification.

None of these approaches is without trade-offs: use of a separate mask array requires allocation of an additional Boolean array, which adds overhead in both storage and computation. A sentinel value reduces the range of valid values that can be represented, and may require extra (often non-optimized) logic in CPU and GPU arithmetic. Common special values like NaN are not available for all data types.

As in most cases where no universally optimal choice exists, different languages and systems use different conventions. For example, the R language uses reserved bit patterns within each data type as sentinel values indicating missing data, while the SciDB system uses an extra byte attached to every cell which indicates a NA state.

Missing Data in Pandas

The way in which Pandas handles missing values is constrained by its reliance on the NumPy package, which does not have a built-in notion of NA values for non-floating-point data types.

Pandas could have followed R's lead in specifying bit patterns for each individual data type to indicate nullness, but this approach turns out to be rather unwieldy. While R contains four basic data types, NumPy supports *far* more than this: for example, while R has a single integer type, NumPy supports *fourteen* basic integer types once you account for available precisions, signedness, and endianness of the encoding. Reserving a specific bit pattern in all available NumPy types would lead to an unwieldy amount of overhead in special-casing various operations for various types, likely even requiring a new fork of the NumPy package. Further, for the smaller data types (such as 8-bit integers), sacrificing a bit to use as a mask will significantly reduce the range of values it can represent.

NumPy does have support for masked arrays – that is, arrays that have a separate Boolean mask array attached for marking data as "good" or "bad." Pandas could have derived from this, but the overhead in both storage, computation, and code maintenance makes that an unattractive choice.

With these constraints in mind, Pandas chose to use sentinels for missing data, and further chose to use two already-existing Python null values: the special floating-point `NaN` value, and the Python `None` object. This choice has some side effects, as we will see, but in practice ends up being a good compromise in most cases of interest.

`None` : Pythonic missing data

The first sentinel value used by Pandas is `None` , a Python singleton object that is often used for missing data in Python code. Because it is a Python object, `None` cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type `'object'` (i.e., arrays of Python objects):

In [127]:

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

Out[127]:

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

In [128]:

```
vals1.sum()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-128-30a3fc8c6726> in <module>()
----> 1 vals1.sum()

~\Anaconda3\lib\site-packages\numpy\core\_methods.py in _sum(a, axis, dtype,
 out, keepdims)
     30 
     31 def _sum(a, axis=None, dtype=None, out=None, keepdims=False):
---> 32     return umr_sum(a, axis, dtype, out, keepdims)
     33 
     34 def _prod(a, axis=None, dtype=None, out=None, keepdims=False):

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

`NaN` : Missing numerical data

The other missing data representation, `NaN` (acronym for *Not a Number*), is different; it is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation:

In [ ]:

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

In [ ]:

```
1 + np.nan
```

In [ ]:

```
1*np.nan
```

In [ ]:

```
vals2.sum(), vals2.min(), vals2.max()
```

In [ ]:

```
np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```

NaN and None in Pandas

`NaN` and `None` both have their place, and Pandas is built to handle the two of them nearly interchangeably, converting between them where appropriate:

In [ ]:

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

For types that don't have an 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:

In [ ]:

```
x = pd.Series(range(2), dtype=int)
x
```

In [ ]:

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

Operating on Null Values

As we have seen, Pandas treats `None` and `NaN` as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:

- `isnull()` : Generate a boolean mask indicating missing values
- `notnull()` : Opposite of `isnull()`

- `dropna()` : Return a filtered version of the data
- `fillna()` : Return a copy of the data with missing values filled or imputed

Detecting null values

Pandas data structures have two useful methods for detecting null data: `isnull()` and `notnull()` . Either one will return a Boolean mask over the data. For example:

In [ ]:

```
data = pd.Series([1, np.nan, 'hello', None])
```

In [ ]:

```
data[data.notnull()]
```

Dropping null values

In addition to the masking used before, there are the convenience methods, `dropna()` (which removes NA values) and `fillna()` (which fills in NA values). For a `Series` , the result is straightforward:

In [ ]:

```
data.dropna()
```

In [ ]:

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

We cannot drop single values from a `DataFrame` ; we can only drop full rows or full columns. Depending on the application, you might want one or the other, so `dropna()` gives a number of options for a `DataFrame` .

By default, `dropna()` will drop all rows in which *any* null value is present:

In [ ]:

```
df.dropna()
```

In [ ]:

```
df.dropna(axis='columns')
```

Filling null values

Sometimes rather than dropping NA values, you'd rather replace them with a valid value. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. You could do this in-place using the `isnull()` method as a mask, but because it is such a common operation Pandas provides the `fillna()` method, which returns a copy of the array with the null values replaced.

Consider the following `Series` :

In [ ]:

```
data = pd.Series([1, np.nan, 2, None, 3], index=['a', 'b', 'c', 'd', 'e'])
data
```

In [ ]:

```
data.fillna(0)
```

In [ ]:

```
# forward-fill
data.fillna(method='ffill')
```

In [ ]:

```
# back-fill
data.fillna(method='bfill')
```

In [ ]:

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

Combining Datasets: Concat and Append

Some of the most interesting studies of data come from combining different data sources. These operations can involve anything from very straightforward concatenation of two different datasets, to more complicated database-style joins and merges that correctly handle any overlaps between the datasets. `Series` and `DataFrame` s are built with this type of operation in mind, and Pandas includes functions and methods that make this sort of data wrangling fast and straightforward.

Here we'll take a look at simple concatenation of `Series` and `DataFrame` s with the `pd.concat` function; later we'll dive into more sophisticated in-memory merges and joins implemented in Pandas.

In [ ]:

```
def make_df(cols, ind):
    """Quickly make a DataFrame"""
    data = {c: [str(c) + str(i) for i in ind]
            for c in cols}
    return pd.DataFrame(data, ind)

# example DataFrame
make_df('ABC', range(3))
```

Simple Concatenation with `pd.concat`

Pandas has a function, pd.concat(), which has a similar syntax to np.concatenate but contains a number of options that we'll discuss momentarily:

Signature in Pandas v0.18

pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False, copy=True) pd.concat() can be used for a simple concatenation of Series or DataFrame objects, just as np.concatenate() can be used for simple concatenations of arrays:

In [ ]:

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
```

By default, the concatenation takes place row-wise within the `DataFrame` (i.e., `axis=0`). Like `np.concatenate` , `pd.concat` allows specification of an axis along which concatenation will take place.

Aggregation and Grouping

Simple Aggregation in Pandas

Earlier, we explored some of the data aggregations available for NumPy arrays. As with a one-dimensional NumPy array, for a Pandas `Series` the aggregates return a single value:

In [ ]:

```
rng = np.random.RandomState(42)
ser = pd.Series(rng.rand(5))
ser
```

In [ ]:

```
print(ser.sum())
print(ser.mean())
```

For a `DataFrame` , by default the aggregates return results within each column:

In [ ]:

```
df = pd.DataFrame({'A': rng.rand(5),
                   'B': rng.rand(5)})
df
```

In [ ]:

```
df.mean()
```

By specifying the `axis` argument, you can instead aggregate within each row:

In [ ]:

```
df.mean(axis='columns')
```

The following table summarizes some other built-in Pandas aggregations:

Aggregation	Description
`count()`	Total number of items
`first()` , `last()`	First and last item
`mean()` , `median()`	Mean and median
`min()` , `max()`	Minimum and maximum
`std()` , `var()`	Standard deviation and variance
`mad()`	Mean absolute deviation
`prod()`	Product of all items
`sum()`	Sum of all items

These are all methods of `DataFrame` and `Series` objects.

With this we have completed some useful methods of pandas. But that's not it... There's more to pandas. For the extra content we encourage you to visit pandas documentation

Pandas online documentation (http://pandas.pydata.org/)

Matplotlib

Visualization with Matplotlib

We'll now take an in-depth look at the Matplotlib package for visualization in Python. Matplotlib is a multi-platform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack. It was conceived by John Hunter in 2002, originally as a patch to IPython for enabling interactive MATLAB-style plotting via gnuplot from the IPython command line. IPython's creator, Fernando Perez, was at the time scrambling to finish his PhD, and let John know he wouldn’t have time to review the patch for several months. John took this as a cue to set out on his own, and the Matplotlib package was born, with version 0.1 released in 2003. It received an early boost when it was adopted as the plotting package of choice of the Space Telescope Science Institute (the folks behind the Hubble Telescope), which financially supported Matplotlib’s development and greatly expanded its capabilities.

One of Matplotlib’s most important features is its ability to play well with many operating systems and graphics backends. Matplotlib supports dozens of backends and output types, which means you can count on it to work regardless of which operating system you are using or which output format you wish. This cross-platform, everything-to-everyone approach has been one of the great strengths of Matplotlib. It has led to a large user base, which in turn has led to an active developer base and Matplotlib’s powerful tools and ubiquity within the scientific Python world.

Importing Matplotlib

Just as we use the `np` shorthand for NumPy and the `pd` shorthand for Pandas, we will use some standard shorthands for Matplotlib imports:

In [ ]:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Setting Styles

We will use the `plt.style` directive to choose appropriate aesthetic styles for our figures. Here we will set the `classic` style, which ensures that the plots we create use the classic Matplotlib style:

In [ ]:

```
plt.style.use('classic')
```

Plotting from an IPython notebook

The IPython notebook is a browser-based interactive data analysis tool that can combine narrative, code, graphics, HTML elements, and much more into a single executable document (see IPython: Beyond Normal Python (01.00-IPython-Beyond-Normal-Python.ipynb)).

Plotting interactively within an IPython notebook can be done with the `%matplotlib` command, and works in a similar way to the IPython shell. In the IPython notebook, you also have the option of embedding graphics directly in the notebook, with two possible options:

- `%matplotlib notebook` will lead to *interactive* plots embedded within the notebook
- `%matplotlib inline` will lead to *static* images of your plot embedded in the notebook

For this book, we will generally opt for `%matplotlib inline` :

In [ ]:

```
import numpy as np
x = np.linspace(0, 10, 100)

fig = plt.figure()
plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--');
```

Saving Figures to File

One nice feature of Matplotlib is the ability to save figures in a wide variety of formats. Saving a figure can be done using the `savefig()` command. For example, to save the previous figure as a PNG file, you can run this:

In [ ]:

```
fig.savefig('my_figure.png')
```

In [ ]:

```
from IPython.display import Image
Image('my_figure.png') #This confirms that the image has been saved!
```

Two Interfaces for the Price of One

A potentially confusing feature of Matplotlib is its dual interfaces: a convenient MATLAB-style state-based interface, and a more powerful object-oriented interface. We'll quickly highlight the differences between the two here.

MATLAB-style Interface

Matplotlib was originally written as a Python alternative for MATLAB users, and much of its syntax reflects that fact. The MATLAB-style tools are contained in the pyplot (`plt`) interface. For example, the following code will probably look quite familiar to MATLAB users:

In [ ]:

```
plt.figure()  # create a plot figure

# create the first of two panels and set current axis
plt.subplot(2, 1, 1) # (rows, columns, panel number)
plt.plot(x, np.sin(x))

# create the second panel and set current axis
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x));
```

Object-oriented interface

The object-oriented interface is available for these more complicated situations, and for when you want more control over your figure. Rather than depending on some notion of an "active" figure or axes, in the object-oriented interface the plotting functions are methods of explicit Figure and Axes objects. To re-create the previous plot using this style of plotting, you might do the following:

In [ ]:

```
# First create a grid of plots
# ax will be an array of two Axes objects
fig, ax = plt.subplots(2)

# Call plot() method on the appropriate object
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x));
```

In [ ]:

```
%matplotlib inline
plt.style.use('seaborn-whitegrid')
```

In [ ]:

```
ax = plt.axes() # For a grid display
```

In [ ]:

```
ax = plt.axes()
x = np.linspace(0,10,1000)
ax.plot(x,np.sin(x))
```

In [ ]:

```
plt.plot(x,np.sin(x))
plt.plot(x,np.cos(x))
```

Adjusting the Plot: Line Colors and Styles

The first adjustment you might wish to make to a plot is to control the line colors and styles. The `plt.plot()` function takes additional arguments that can be used to specify these. To adjust the color, you can use the `color` keyword, which accepts a string argument representing virtually any imaginable color. The color can be specified in a variety of ways:

In [ ]:

```
plt.plot(x, np.sin(x - 0), color='blue')        # specify color by name
plt.plot(x, np.sin(x - 1), color='g')           # short color code (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75')        # Grayscale between 0 and 1
plt.plot(x, np.sin(x - 3), color='#FFDD44')     # Hex code (RRGGBB from 00 to FF)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB tuple, values 0 to 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); # all HTML color names supported
```

Adjusting the Plot: Axes Limits

Matplotlib does a decent job of choosing default axes limits for your plot, but sometimes it's nice to have finer control. The most basic way to adjust axis limits is to use the `plt.xlim()` and `plt.ylim()` methods:

In [ ]:

```
plt.plot(x, np.sin(x))

plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5);
```

In [ ]:

```
plt.plot(x, np.sin(x))
plt.axis('tight');
```

In [ ]:

```
plt.plot(x, np.sin(x))
plt.axis('equal');
```

Labeling Plots

As the last piece of this section, we'll briefly look at the labeling of plots: titles, axis labels, and simple legends.

Titles and axis labels are the simplest such labels—there are methods that can be used to quickly set them:

In [ ]:

```
plt.plot(x, np.sin(x))
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
```

When multiple lines are being shown within a single axes, it can be useful to create a plot legend that labels each line type. Again, Matplotlib has a built-in way of quickly creating such a legend. It is done via the (you guessed it) `plt.legend()` method. Though there are several valid ways of using this, I find it easiest to specify the label of each line using the `label` keyword of the plot function:

In [ ]:

```
plt.plot(x, np.sin(x), '-g', label='sin(x)')
plt.plot(x, np.cos(x), ':b', label='cos(x)')
plt.axis('equal')

plt.legend();
```

Scatter Plots with `plt.plot`

In the previous section we looked at `plt.plot` / `ax.plot` to produce line plots. It turns out that this same function can produce scatter plots as well:

In [ ]:

```
x = np.linspace(0, 10, 30)
y = np.sin(x)
z = np.cos(x)
plt.plot(x, y, 'o', color='red')
plt.plot(x, z, 'x', color='blue')
```

Scatter Plots with `plt.scatter`

A second, more powerful method of creating scatter plots is the `plt.scatter` function, which can be used very similarly to the `plt.plot` function:

In [ ]:

```
plt.scatter(x, y, marker='o')
```

The primary difference of `plt.scatter` from `plt.plot` is that it can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.

Let's show this by creating a random scatter plot with points of many colors and sizes. In order to better see the overlapping results, we'll also use the `alpha` keyword to adjust the transparency level:

In [ ]:

```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,
            cmap='winter')
plt.colorbar();  # show color scale
```

For more cmap types visit the following link: https://matplotlib.org/examples/color/colormaps_reference.html (https://matplotlib.org/examples/color/colormaps_reference.html)

`plot` Versus `scatter`: A Note on Efficiency

Aside from the different features available in `plt.plot` and `plt.scatter` , why might you choose to use one over the other? While it doesn't matter as much for small amounts of data, as datasets get larger than a few thousand points, `plt.plot` can be noticeably more efficient than `plt.scatter` . The reason is that `plt.scatter` has the capability to render a different size and/or color for each point, so the renderer must do the extra work of constructing each point individually. In `plt.plot` , on the other hand, the points are always essentially clones of each other, so the work of determining the appearance of the points is done only once for the entire set of data. For large datasets, the difference between these two can lead to vastly different performance, and for this reason, `plt.plot` should be preferred over `plt.scatter` for large datasets.

Histograms, Binnings, and Density

A simple histogram can be a great first step in understanding a dataset. Earlier, we saw a preview of Matplotlib's histogram function which creates a basic histogram in one line, once the normal boiler-plate imports are done:

In [ ]:

```
data = np.random.randn(1000)
plt.hist(data)
```

In [ ]:

```
plt.hist(data, bins=30, normed=True, alpha=0.5,
         histtype='stepfilled', color='steelblue',
         edgecolor='black');
```

In [ ]:

```

```