DATA SOCIETY®

Introduction to Python - Day 3

"One should look for what is and not what he thinks should be."
-Albert Einstein.

Module completion checklist

Objective	Complete
Work with numpy objects	
Summarize use cases of pandas and update directory settings	
Demonstrate use of basic operations on series	
Demonstrate use of basic operations on dataframes	
Load data into Python using pandas	
Summarize data using pandas	

Data wrangling and exploration

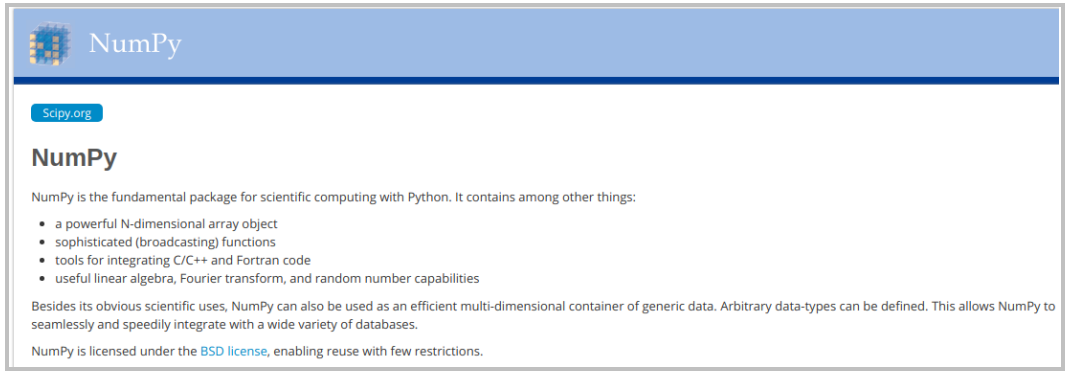
- Remember, a data scientist must be able to:

1. **Wrangle** the data (gather, clean, and sample data to get a suitable dataset)
2. **Manage** the data for easy access by the organization
3. **Explore** the data to generate a hypothesis

- Today, we will learn how to use two powerful Python libraries, **NumPy and Pandas**, that will help us achieve these goals!

Introduction to NumPy

- NumPy is widely used in machine learning and scientific computing due to its basic core data structure: array
- It is also widely used in combination with `matplotlib` and other plotting libraries to create graphs
- NumPy's array functions are similar to those available for vectors in Matlab and R



Creating arrays

- There are multiple ways to create a numpy array
- One of the easiest is to make it from a `list` and using NumPy's `array()` function
- To use the `array()` function, we need to import `numpy`
- Once again, when writing code, we usually want to **import all packages needed for the program at the beginning**
- However, **since we are learning as we go, we import them as we learn in class**

```
# Import numpy as 'np' sets 'np' as the shortcut/alias.
import numpy as np

# Create an array from a list.
arr = np.array([17, -10, 16.8, 11])
print(arr)

# Check the type of the object.
```

```
[ 17.  -10.   16.8  11. ]
```

```
print(type(arr))
```

```
<class 'numpy.ndarray'>
```

Dtype in arrays

- NumPy arrays have a property of `dtype` which records the data type of the array's members
- NumPy arrays are **required to have the same data type**, that is why they are called `atomic` data structures (i.e. structures that allow a single data type)!

```
# Check the data type stored in the array.
print(arr.dtype)
```

```
float64
```

Using ndarray

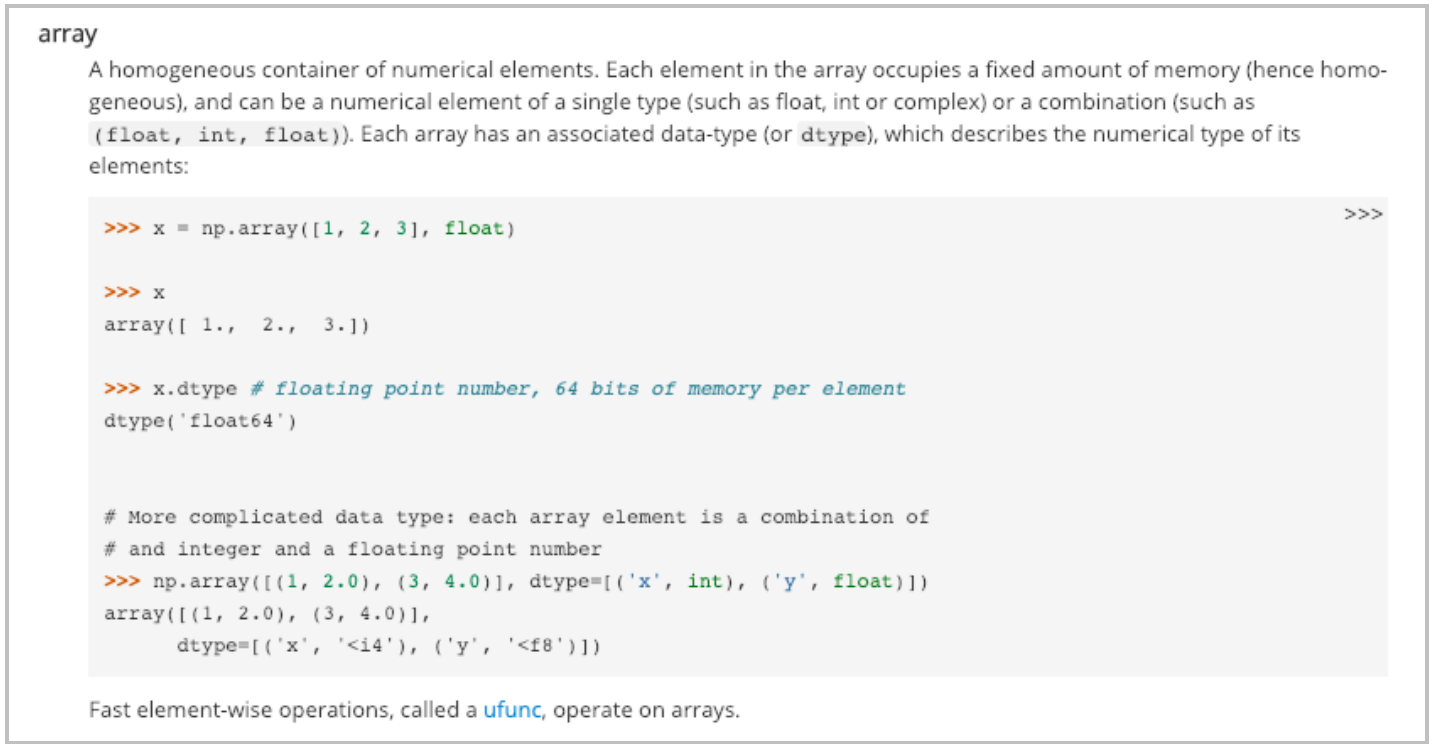
- The most important data type that NumPy provides is the "N-dimensional array," `ndarray`
- An `ndarray` is similar to a Python list in which all members have the same data type
- We create it using `np.array()`

```
x = np.array([3, 19, 7, 11])
print(x)
```

```
[ 3 19  7 11]
```

Documentation for ndarray

- Each package in Python, like `numpy` and `pandas` (which we will learn about later today), has ***documentation*** for each function within



array

A homogeneous container of numerical elements. Each element in the array occupies a fixed amount of memory (hence homogeneous), and can be a numerical element of a single type (such as float, int or complex) or a combination (such as `(float, int, float)`). Each array has an associated data-type (or `dtype`), which describes the numerical type of its elements:

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

>>> x
array([ 1.,  2.,  3.])

>>> x.dtype # floating point number, 64 bits of memory per element
dtype('float64')


# More complicated data type: each array element is a combination of
# and integer and a floating point number
>>> np.array([(1, 2.0), (3, 4.0)], dtype=[('x', int), ('y', float)])
array([(1, 2.0), (3, 4.0)],
      dtype=[('x', '<i4'), ('y', '<f8')])
```

Fast element-wise operations, called a ufunc, operate on arrays.

Building an array with linspace

- Another function we can use to build an array is `np.linspace`

```
y = np.linspace(-2, -1, 25)
print(y)
```

```
[-2.         -1.95833333 -1.91666667 -1.875
-1.83333333 -1.79166667
 -1.75       -1.70833333 -1.66666667 -1.625
-1.58333333 -1.54166667
 -1.5        -1.45833333 -1.41666667 -1.375
-1.33333333 -1.29166667
 -1.25       -1.20833333 -1.16666667 -1.125
-1.08333333 -1.04166667
 -1.        ]
```

- This function will return 25 numbers between -2 and -1

numpy.linspace

numpy.linspace(*start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0*) [source]

Return evenly spaced numbers over a specified interval.

Returns *num* evenly spaced samples, calculated over the interval [*start, stop*].

The endpoint of the interval can optionally be excluded.

Changed in version 1.16.0: Non-scalar *start* and *stop* are now supported.

Alternative ways of accessing functions

- Another way, which can be useful if you are only going to use a handful of functions from a library, is as follows:

```
from numpy import array, linspace
x = array([0.01, 0.45, -0.3])
y = linspace(0, 1, 50)
```

- With this syntax, we can use array or linspace without the np. prefix

NumPy array 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)

NumPy array data types (cont'd)

Data type	Description
“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_”	Byte (-128 to 127)
Shorthand for “float64”	Integer (-32768 to 32767)
“float16”	Integer (-2147483648 to 2147483647)
“int64”	Integer (-9223372036854775808 to 9223372036854775807)

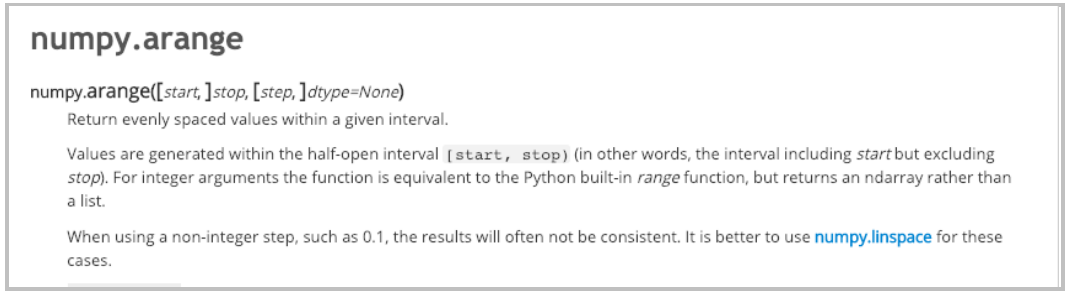
Arrays from sequences

- We can also create an array that contains a sequence of numbers
- To create the range of numbers of 0 to 50, use the `arange` command

```
rng = np.arange(0, 51)
print(rng)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47
 48 49 50]
```

- The last number in the range is one less than the value you provided, so we provide 51 to ensure that the last value is 50



numpy.arange

numpy.arange([*start*,]*stop*, [*step*,]*dtype=None*)

Return evenly spaced values within a given interval.

Values are generated within the half-open interval `[start, stop)` (in other words, the interval including *start* but excluding *stop*). For integer arguments the function is equivalent to the Python built-in *range* function, but returns an ndarray rather than a list.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use numpy.linspace for these cases.

Arrays from sequences - using a step size

- We can also have the numbers increase by a step size other than 1

```
evens = np.arange(0, 23, 2)
print(evens)
```

```
[ 0  2  4  6  8 10 12 14 16 18 20 22]
```

```
quarters = np.arange(0, 1, .25)  #<- contains 0 to 0.75
print(quarters)
```

```
[0.   0.25 0.5  0.75]
```

Helper functions: min and max

- Arrays have many useful functions available
- For instance, for numeric arrays, you can check its maximum or minimum value, or the sum of its elements

numpy.amin¶

numpy.amin(*a, axis=None, out=None, keepdims=<no value>, initial=<no value>, where=<no value>*) [source]

Return the minimum of an array or minimum along an axis.

numpy.amax

numpy.amax(*a, axis=None, out=None, keepdims=<no value>, initial=<no value>, where=<no value>*) [source]

Return the maximum of an array or maximum along an axis.

```
# Generate 5 numbers between 15 and 19.
x = np.linspace(15, 19, 5)

# Find the min of x.
np.amin(x)
```

```
15.0
```

```
# Find the max of x.
np.amax(x)
```

```
19.0
```

Convert an array to a list

- We can convert an array to a normal `list` with the `list` function
- We will demonstrate that with the array we created earlier, `evens`

```
print(list(evens))
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22]
```

Operations on arrays

- Numeric arrays of the same length can be added, subtracted, multiplied or divided

```
# Save two arrays as variables.
a = np.array([1,1,1,1])
b = np.array([2,2,2,2])

# Addition of arrays.
print(a + b)
```

```
[3 3 3 3]
```

```
# Subtraction of arrays.
print(a - b)
```

```
[-1 -1 -1 -1]
```

```
# Multiplication of arrays.
print(a * b)
```

```
[2 2 2 2]
```

```
# Division of arrays.
print(a / b)
```

```
[0.5 0.5 0.5 0.5]
```

- In NumPy, these operations are defined **element-wise**
- In other words, each pair of corresponding elements in the two arrays are operated on, and the result is a new array containing each result

Mathematical functions on lists

- You might be wondering if we can perform operations on lists, the answer is **no**
- If we wanted an absolute value of a list of numbers, we **can't** do this:

```
abs([-2, -7, 1])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-55-e2459d669344> in <module>()
----> 1 abs([-2, -7, 1])

TypeError: bad operand type for abs(): 'list'
```

- The `TypeError` tells us that abs is not set up to handle lists!

Mathematical functions on arrays

- Remember when we transformed a list into a numpy array?
- Many functions in NumPy are **vectorized** functions, meaning they can handle a single input or an array of inputs
- When we use the same function `abs()` on an `np.` object, we see different results

```
print(np.abs(-3))
```

```
3
```

```
print(np.abs([-2, -7, 1]))
```

```
[2 7 1]
```

Accessing array values

- Just like with lists, we can grab individual elements or a range of elements from an array using square bracket notation

```
nums = np.arange(20, 30, .5)
print(len(nums))
```

```
20
```

```
print(nums[1])  #<- get the second element
```

```
20.5
```

```
print(nums[0:3]) #<- get the first three elements
```

```
[20.  20.5 21. ]
```

Logical filtering

- You can't filter lists by a logical condition, however **you can filter arrays by a logical condition**
- If the corresponding condition is met, then it retains the value from the array, otherwise it excludes it

```
print(nums)
```

```
[20.  20.5 21.  21.5 22.  22.5 23.  23.5 24.  24.5 25.  25.5 26.  26.5
 27.  27.5 28.  28.5 29.  29.5]
```

```
large_nums = nums[nums > 26]
print(large_nums)
```

```
[26.5 27.  27.5 28.  28.5 29.  29.5]
```

Logical filtering (cont'd)

```
print(nums)
```

```
[20.  20.5 21.  21.5 22.  22.5 23.  23.5 24.  24.5 25.  25.5 26.  26.5
 27.  27.5 28.  28.5 29.  29.5]
```

```
large_nums = nums[nums > 26]
print(large_nums)
```

```
[26.5 27.  27.5 28.  28.5 29.  29.5]
```

- It is important to remember that there are a few steps happening here:
  - The expression within the brackets produces a so-called **Boolean mask**: an array of `True/False` values
  - The logical statement `> 26`, is applied to each value of `nums`, so the result is an array of `True/False` values
  - Our `nums` array and the mask array are then lined up, and the values out of `nums` are filtered based on the corresponding mask value

Two-dimensional arrays

- As the name suggests, `ndarray` (i.e. n-dimensional array) can have more than one dimension!
- Multiple dimensions are created by nesting lists within each other
- To create a 2D array (a matrix), we can write the following:

```
mat = np.array([
        [8, 2, 6, 8],
        [4, 5, 7, 2],
        [3, 9, 7, 1]
       ])
print(mat)
```

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

Two-dimensional arrays - shape

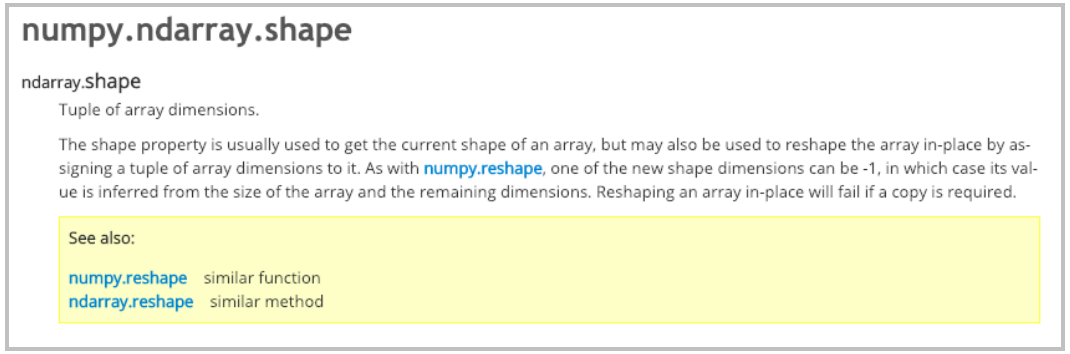
- The `shape` property of an array tells us the size of each of its dimensions

```
print(mat.shape) #<- 3 rows and 4 columns -- returned as a tuple
```

```
(3, 4)
```

```
nrows, ncols = mat.shape
print(nrows)
```

```
3
```



numpy.ndarray.shape

ndarray.shape

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with numpy.reshape, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

See also:

numpy.reshape similar function
ndarray.reshape similar method

Two-dimensional arrays - extracting elements

- To extract a value from the matrix, we use 2-dimensional bracket notation:
  - **1st number is the row position**
  - **2nd is the column position**

```
print(mat[1, 3]) #<- 2nd row 4th column - remember that indexing starts at 0!
```

```
2
```

Two-dimensional arrays - rows

- To extract an entire row of a matrix, replace the column ID with colon
- The colon indicates you want all of the columns
- Alternatively, you can specify a range of column positions, which uses normal Python list slicing notation

```
print(mat[0, :]) #<- first row
```

```
[8 2 6 8]
```

```
print(mat[0, 0:2]) #<- first row and just first 2 columns
```

```
[8 2]
```

Two-dimensional arrays - columns

- Similarly, to extract a single column, replace the row argument with a colon or leave it blank

```
print(mat[:, 2]) #<- 3rd column
```

```
[6 7 7]
```

```
print(mat[1:3, 2]) #<- 3rd column but skipping over the first row
```

```
[7 7]
```

```
print(mat[1:3, 2:3]) #<- same as previous, but maintains the vertical structure of the column
```

```
[[7]
 [7]]
```

Knowledge check 1

Exercise 1

Module completion checklist

Objective	Complete
Work with numpy objects	✔
Summarize use cases of pandas and update directory settings	
Demonstrate use of basic operations on series	
Demonstrate use of basic operations on dataframes	
Load data into Python using pandas	
Summarize data using pandas	

Dataset manipulation with Pandas

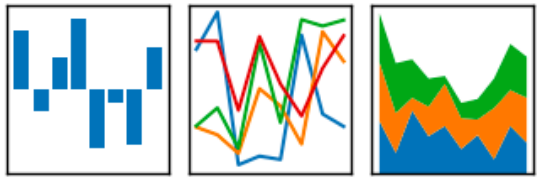
- `Pandas` is a powerful library for **cleaning and analyzing datasets in Python**
- We learned about `numpy`, which helps us work with datasets, specifically arrays of numbers, to get ready for machine learning
- `Pandas` will help us cleaning and analyzing datasets of all kinds
- For complete documentation, ***click here***

A little more about Pandas

- Pandas is an effective tool to **read, write and manipulate data**
- Pandas contains tools to perform high-performance **merging and joining datasets**
- Pandas is **highly optimized for performance**, with critical code paths written in C

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Import Pandas and os

- Let's import the `pandas` library
- Note: it is not required that you also import `numpy` in order to use `pandas`
- However, you will often see both of them imported since many projects make use of both

```
import pandas as pd
```

- We now are going to introduce a package that allows you to set your working directory
- This will be the directory where your data lies, allowing you to import data directly from there

```
import os
```

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- Let the `main_dir` be the variable corresponding to your `af-werx` folder

```
# Set `main_dir` to the location of your `af-werx` folder (for Linux).
main_dir = "/home/[username]/af-werx"
```

```
# Set `main_dir` to the location of your `af-werx` folder (for Mac).
main_dir = '/Users/[username]/af-werx'
```

```
# Set `main_dir` to the location of your `af-werx` folder (for Windows).
main_dir = "C:\\Users\\[username]\\af-werx"
```

```
# Make `data_dir` from the `main_dir` and
# remainder of the path to data directory.
data_dir = main_dir + "/data"
```

Working directory

- Set working directory to the `data_dir` variable we set
- We do this using the `os.chdir` function, change directory
- We can then check the working directory using `.getcwd()`
- For complete documentation of the `os` package, ***click here***

```
# Set working directory.
os.chdir(data_dir)
```

```
# Check working directory.
print(os.getcwd())
```

```
/home/[user-name]/af-werx/data
```

Module completion checklist

Objective	Complete
Work with numpy objects	✔
Summarize use cases of pandas and update directory settings	✔
Demonstrate use of basic operations on series	
Demonstrate use of basic operations on dataframes	
Load data into Python using pandas	
Summarize data using pandas	

Series

- The first `pandas` object we'll learn about is a `Series`
- Think of `Series` as a NumPy array but with ***many additional properties and methods***
- We can create `Series` from a normal Python `list`

```
num_series = pd.Series([45, 89, 67, 33])
print(num_series)
```

```
0    45
1    89
2    67
3    33
dtype: int64
```

- In fact, the values are stored in an `ndarray`!
- To extract just the values as an `ndarray`, use the `.values` property of `Series`

```
print(num_series.values)
```

```
[45 89 67 33]
```

Date series: ranges by month

- `pandas` supports series of dates, making it a great choice for time series analysis
- Date series can be created in a couple ways

```
# Go in intervals of month.
print(pd.date_range(start = '20170101', end = '20170331', freq = 'M'))
```

```
DatetimeIndex(['2017-01-31', '2017-02-28', '2017-03-31'], dtype='datetime64[ns]', freq='M')
```

```
# Not specifying end, but instead the start, freq, and how many periods.
print(pd.date_range(start = '20170101', freq = 'M', periods = 4))
```

```
DatetimeIndex(['2017-01-31', '2017-02-28', '2017-03-31', '2017-04-30'], dtype='datetime64[ns]',
freq='M')
```

Date series: ranges by hour

- This function can also create hourly series

```
print(pd.date_range(start = '20170101', end = '20170102', freq = 'H'))
```

```
DatetimeIndex(['2017-01-01 00:00:00', '2017-01-01 01:00:00',
               '2017-01-01 02:00:00', '2017-01-01 03:00:00',
               '2017-01-01 04:00:00', '2017-01-01 05:00:00',
               '2017-01-01 06:00:00', '2017-01-01 07:00:00',
               '2017-01-01 08:00:00', '2017-01-01 09:00:00',
               '2017-01-01 10:00:00', '2017-01-01 11:00:00',
               '2017-01-01 12:00:00', '2017-01-01 13:00:00',
               '2017-01-01 14:00:00', '2017-01-01 15:00:00',
               '2017-01-01 16:00:00', '2017-01-01 17:00:00',
               '2017-01-01 18:00:00', '2017-01-01 19:00:00',
               '2017-01-01 20:00:00', '2017-01-01 21:00:00',
               '2017-01-01 22:00:00', '2017-01-01 23:00:00',
               '2017-01-02 00:00:00'],
              dtype='datetime64[ns]', freq='H')
```

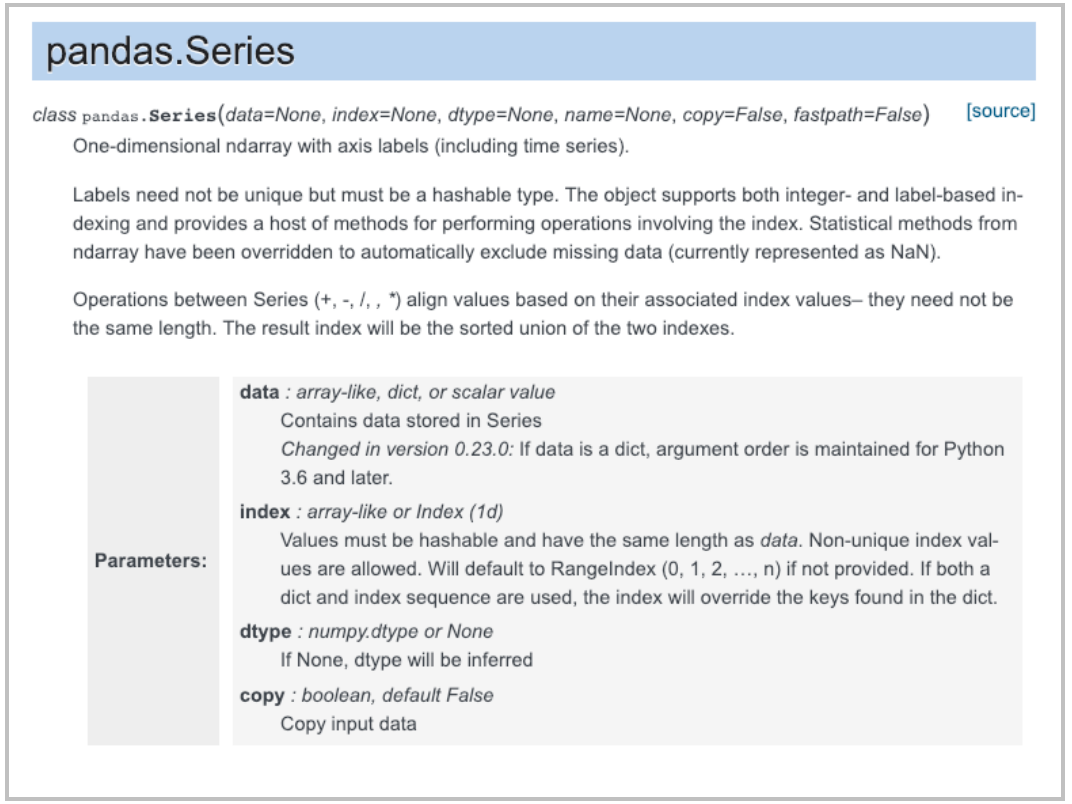
- You can create series by year, by minute, by second, without needing a date
- Many formats are available!

Series methods

- `Series` are more powerful than base Python lists due to the additional attributes and methods they possess

```
norm_series = pd.Series(np.arange(5, 20, 5))
print(norm_series)
```

```
0     5
1    10
2    15
dtype: int64
```



pandas.Series

class `pandas.`**`Series`**(*data=None, index=None, dtype=None, name=None, copy=False, fastpath=False*) [source]

One-dimensional ndarray with axis labels (including time series).

Labels need not be unique but must be a hashable type. The object supports both integer- and label-based indexing and provides a host of methods for performing operations involving the index. Statistical methods from ndarray have been overridden to automatically exclude missing data (currently represented as NaN).

Operations between Series (+, -, /, , *) align values based on their associated index values– they need not be the same length. The result index will be the sorted union of the two indexes.

Parameters:

data *: array-like, dict, or scalar value*
Contains data stored in Series
Changed in version 0.23.0: If data is a dict, argument order is maintained for Python 3.6 and later.

index *: array-like or Index (1d)*
Values must be hashable and have the same length as *data*. Non-unique index values are allowed. Will default to RangeIndex (0, 1, 2, …, n) if not provided. If both a dict and index sequence are used, the index will override the keys found in the dict.

dtype *: numpy.dtype or None*
If None, dtype will be inferred

copy *: boolean, default False*
Copy input data

Series - functions

- Now let's apply some mathematical functions to this series

```
print(norm_series.shape)    #<- number of rows and columns
```

```
(3,)
```

```
print(norm_series.mean())   #<- series mean
```

```
10.0
```

```
print(norm_series.median()) #<- series median
```

```
10.0
```

```
print(norm_series.std())    #<- series std deviation
```

```
5.0
```

Series - functions

- Here are some ways to count items in a series

```
# Show only unique values.
print(norm_series.unique())
```

```
[ 5 10 15]
```

```
# Show number of unique values.
print(norm_series.nunique())
```

```
3
```

```
# Show counts of unique values.
print(norm_series.value_counts())
```

```
15    1
10    1
5     1
dtype: int64
```

```
# Position of the min value.
print(norm_series.idxmin())
```

```
0
```

```
# Position of the max value.
print(norm_series.idxmax())
```

```
2
```

Series - rank

- We can rank items in a series, in ascending order:

```
# Ranks from smallest to largest.
print(norm_series.rank())
```

```
0    1.0
1    2.0
2    3.0
dtype: float64
```

- And in descending order:

```
# Ranks from largest to smallest.
print(norm_series.rank(ascending = False))
```

```
0    3.0
1    2.0
2    1.0
dtype: float64
```

Series - sort and cumulative sum

- We can sort series:

```
# Sorts values.
print(norm_series.sort_values())
```

```
0     5
1    10
2    15
dtype: int64
```

- And find the cumulative sum:

```
# Returns a series that is the cumulative sum of
`norm_series`.
print(norm_series.cumsum())
```

```
0     5
1    15
2    30
dtype: int64
```

Knowledge check 2

Exercise 2

Module completion checklist

Objective	Complete
Work with numpy objects	✔
Summarize use cases of pandas and update directory settings	✔
Demonstrate use of basic operations on series	✔
Demonstrate use of basic operations on dataframes	
Load data into Python using pandas	
Summarize data using pandas	

Dataframes

- Now that we have reviewed `Series`, let's look at what a `dataframe` is
- **A `dataframe` is the single most important object in `pandas`**
  - It is a collection of `series` of equal lengths
  - Just like `series`, dataframes come with many useful methods
- Review complete documentation of the `DataFrame` function ***here***
- For this simple example, we'll build a dataframe using one series similar to what we just built, `Timestamp`
- The second series we will be a set of numbers representing the average number of days people were out of office `OOO`

Series to dataframe

- We create a `dataframe` object with the `pd.dataframe` function, and we specify the `Series` we want to include (in this case, it's `times` and `days` out of office)
- We are going to create two `series`:
  - Our first `series` will consist of times
  - We will use the `date_range` method that we just learned about
  - A second `series` will be made of the average number of days people were out of office, constructed from a list of numbers

```
# Series 1 - times:
times = pd.date_range(start = '20170101', end = '20170630', freq = 'M')

# Series 2 - days out of the office:
days = pd.Series([2, 2, 6, 6, 2, 3])
```

Generate dataframe from series

- Create a `dataframe` using dictionary-like syntax:
  - Dictionary `keys` become column names of the dataframe, and
  - Dictionary `values` become column values
- Inspect the `dataframe` by looking at the first few rows, using `.head()`

```
# Create a dataframe from the two series we just created, as a dictionary.
average_ooo = pd.DataFrame({'Timestamp': times, 'OOO': days})

# View the first few rows of the dataframe, using the pandas function `.head()`.
print(average_ooo.head())
```

```
   Timestamp  OOO
0 2017-01-31    2
1 2017-02-28    2
2 2017-03-31    6
3 2017-04-30    6
4 2017-05-31    2
```

Look-up dataframe information

- As with arrays and lists, we can look up the `type` of the created object as well as its `shape`

```
# Look up the type of object.
print(type(average_ooo))
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
# Look up its shape.
print(average_ooo.shape)
```

```
(6, 2)
```

`DataFrame` is a rectangular object - it will have rows and columns just like a matrix:

1. The first number in parentheses gives us the number of rows, and
2. The second number is the number of columns

Dataframe description metrics

- There are many metrics you can pull from a `DataFrame` object
- We will now review some key metrics that will help us understand our data
  - `.columns` returns columns names
  - `.info()` gives us some extra info about each column like its data type, and how many null values it has
  - `.describe()` computes summary statistics on any numeric column

Dataframe description metrics

- Now, let's preview these metrics on the `sts121` dataset

```
print(average_ooo.columns)
```

```
Index(['Timestamp', 'OOO'], dtype='object')
```

```
print(average_ooo.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 2 columns):
Timestamp    6 non-null datetime64[ns]
OOO          6 non-null int64
dtypes: datetime64[ns](1), int64(1)
memory usage: 176.0 bytes
None
```

```
print(average_ooo.describe())
```

```
              OOO
count  6.000000
mean   3.500000
std    1.974842
min    2.000000
25%    2.000000
50%    2.500000
75%    5.250000
max    6.000000
```

Extracting a single column

- To extract a column, just put its name in quotation marks into square brackets like this: `data_frame['column_name']`

```
print(average_ooo['Timestamp'])
```

```
0   2017-01-31
1   2017-02-28
2   2017-03-31
3   2017-04-30
4   2017-05-31
5   2017-06-30
Name: Timestamp, dtype: datetime64[ns]
```

- The resulting object is a `Series` type
- If you would like to get a `DataFrame` object with a single column, then pass the `list` with a single column name into the square brackets like this: `data_frame[['column_name']]`

Extracting multiple columns

- To extract multiple columns, just pass a list of columns

```
print(average_ooo[['Timestamp', 'OOO']])
```

```
   Timestamp  OOO
0 2017-01-31    2
1 2017-02-28    2
2 2017-03-31    6
3 2017-04-30    6
4 2017-05-31    2
5 2017-06-30    3
```

Extracting a single row

- To extract a particular row from a dataframe, we can use a syntax similar to what we used for `ndarrays`, but with one small change: **we must use the `iloc` method!**

```
june_ooo = average_ooo.iloc[5, :]
print(june_ooo)
```

```
Timestamp    2017-06-30 00:00:00
OOO                            3
Name: 5, dtype: object
```

```
june_ooo = average_ooo.iloc[5] #<- equivalent without the colon
print(june_ooo)
```

```
Timestamp    2017-06-30 00:00:00
OOO                            3
Name: 5, dtype: object
```

Working with dataframe indices

- Dataframes in `pandas` have a property called the `index`
- The `index` serves many purposes and is an important concept to understand within `pandas`
- Some main purposes are:
  - identifying data using known indicators, important for analysis, visualization, and interactive console display
  - enabling automatic and explicit data alignment
  - allowing intuitive getting and setting of subsets of the dataset

Index for our dataset

- The `average_ooo` dataframe has an unlabeled column with the numbers `0` to `5`, this is the `index` of our dataframe
- By default, the `index` is simply the row number (starting with `0`), but it can sometimes make sense to use something more descriptive for the index
- We are going to use `set_index` to set our index in `average_ooo`

```
# Let's use the `Timestamp` column as our new index.
average_ooo = average_ooo.set_index('Timestamp')
print(average_ooo)
```

```
             OOO
Timestamp
2017-01-31     2
2017-02-28     2
2017-03-31     6
2017-04-30     6
2017-05-31     2
2017-06-30     3
```

Looking up by the new index

- Now the rows of our dataframe are indexed by the time stamp and the `Timestamp` column has been removed
- This makes it really easy to look up values corresponding to a particular time stamp
- To do this, we now use the `.loc()` method

```
print(average_ooo.index)
```

```
DatetimeIndex(['2017-01-31', '2017-02-28', '2017-03-31', '2017-04-30',
               '2017-05-31', '2017-06-30'],
              dtype='datetime64[ns]', name='Timestamp', freq=None)
```

```
# Look up a specific row by index.
print(average_ooo.loc['2017-02-28'])
```

```
OOO    2
Name: 2017-02-28 00:00:00, dtype: int64
```

Loc vs. iloc

- Notice we used `loc` not `iloc` like in the first example
- The "`i`" in `iloc` stands for integer
- We can always use `iloc` as well
- As it turns out, the row we wanted was in position 1, so we could also say:

```
print(average_ooo.iloc[1])
```

```
OOO    2
Name: 2017-02-28 00:00:00, dtype: int64
```

Reset the index

- To change the index back to the default, use `.reset_index()`, it will
  - Change the Index back to `0..5`
  - Move the `Timestamp` values back into the dataframe as a column

```
average_ooo = average_ooo.reset_index()
print(average_ooo.index)
```

```
RangeIndex(start=0, stop=6, step=1)
```

```
# You can see that now `Timestamp` is once again a column vs. what it looked like when it was an index.
print(average_ooo)
```

```
   Timestamp  OOO
0 2017-01-31    2
1 2017-02-28    2
2 2017-03-31    6
3 2017-04-30    6
4 2017-05-31    2
5 2017-06-30    3
```

Module completion checklist

Objective	Complete
Work with numpy objects	✔
Summarize use cases of pandas and update directory settings	✔
Demonstrate use of basic operations on series	✔
Demonstrate use of basic operations on dataframes	✔
Load data into Python using pandas	
Summarize data using pandas	

Loading data into Python using pandas

- Now that we know some of the key functions of pandas, we can work with actual datasets
- We will be using two datasets today
- **One dataset in class, to learn the concepts**
  - Costa Rica household poverty data by the Inter-American Development Bank

- **One dataset for our in class exercises**
  - Worldwide tuberculosis estimates by the World Health Organization (WHO)

Reading data from a file

- Your data will most likely be stored either in a database or in a file, **you will need to import it into your environment**
- A common data format used for storing and sharing data is a `csv` file format (i.e. comma separated value)
- `pandas` has a `read_csv` function to import such files
- In your course materials, you should have a `csv` file called `household_poverty.csv` - we will use this dataset to experiment with various dataframe functions
- In addition to csv data, Pandas can read a variety of formats, including Excel, JSON, HTML, Stata, SAS, and even from a SQL connection - the full list of readable and writable file formats is available ***here***.
- **Remember to set your data directory before you begin**
- You *MUST* be pointed to the directory where your data is located

Read data from csv file

- We are now going to use the function `read_csv` to read in our `household_poverty` dataset

```
household_poverty = pd.read_csv('household_poverty.csv')
print(household_poverty.head())
```

```
   male      hh_ID  rooms  ...  water_inside  years_of_schooling  Target
0     1  21eb7fcc1      3  ...             1                  10       4
1     1  0e5d7a658      4  ...             1                  12       4
2     0  2c7317ea8      8  ...             1                  11       4
3     1  2b58d945f      5  ...             1                   9       4
4     0  2b58d945f      5  ...             1                  11       4

[5 rows x 14 columns]
```

Inspect data

- What have we just created?
- Let's inspect

```
print(type(household_poverty))  #<- a pandas dataframe!
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
print(len(household_poverty))   #<- returns the number of rows
```

```
1000
```

```
# You can also save the shape of the dataframe into 2 variables
# (since the returned is a tuple with 2 values).

nrows, ncols = household_poverty.shape
print(nrows)
```

```
1000
```

```
print(ncols)
```

```
14
```

Previewing data - using head method

- We've already used `.head()` command that shows the first few rows
- Let's inspect this data

```
print(household_poverty.head())  #<- pulls the first 5 rows (the default is 5)
```

```
   male      hh_ID  rooms  ...  water_inside  years_of_schooling  Target
0     1  21eb7fcc1      3  ...             1                  10       4
1     1  0e5d7a658      4  ...             1                  12       4
2     0  2c7317ea8      8  ...             1                  11       4
3     1  2b58d945f      5  ...             1                   9       4
4     0  2b58d945f      5  ...             1                  11       4

[5 rows x 14 columns]
```

Previewing data - using head method

- We can specify the number of rows we want to see:

```
print(household_poverty.head(3)) #<- pulls the first 3 rows
```

```
   male      hh_ID  rooms  ...  water_inside  years_of_schooling  Target
0     1  21eb7fcc1      3  ...             1                  10       4
1     1  0e5d7a658      4  ...             1                  12       4
2     0  2c7317ea8      8  ...             1                  11       4

[3 rows x 14 columns]
```

Previewing data - using sample method

- We can view some random rows in the dataframe by using the `.sample()` method

```
print(household_poverty.sample(n = 3))       #<- 3 random rows
```

```
     male      hh_ID  rooms  ...  water_inside  years_of_schooling  Target
863     1  0d2895839      4  ...             1                   9       4
173     0  4b301d9b2      5  ...             1                   6       3
163     0  28ec0c747      7  ...             1                   6       4

[3 rows x 14 columns]
```

Previewing data - using sample method

- We can specify to see a percentage rather than an actual exact number

```
print(household_poverty.sample(frac = .02)) #<- a random 2% of the rows
```

```
     male      hh_ID  rooms  ...  water_inside  years_of_schooling  Target
44      1  a57a1f2f4      8  ...             1                  16       4
186     1  322cefd2f      4  ...             1                   0       2
435     1  4476ccd4c     10  ...             1                  11       4
293     0  35c040720      7  ...             1                  12       4
886     0  615188ceb      5  ...             1                   1       4
271     0  159ea258f      7  ...             1                  11       4
602     0  3641ce2d1      4  ...             1                   3       2
106     0  288b0f0fa      4  ...             1                  11       4
13      0  c51f9c774      4  ...             1                   4       4
596     1  06804be1b      5  ...             1                   5       4
539     0  75f505df4      7  ...             1                   6       4
407     0  5c3f7725d      7  ...             1                   0       3
342     1  de5f39915      5  ...             1                  14       4
65      0  5f7699c70      3  ...             1                   9       4
436     1  4476ccd4c     10  ...             1                   6       4
443     0  4476ccd4c     10  ...             1                   0       4
217     0  aa3814397      5  ...             1                  15       4
560     0  78711dc54      4  ...             1                  17       4
82      1  bcc196e5a      7  ...             1                  15       4
794     1  7b2cce7ab      4  ...             1                   2       4

[20 rows x 14 columns]
```

Reviewing household_poverty data

- We are now going to get to know our data better, using `pandas` techniques we just learned:
  - `.columns`
  - `.dtypes`
  - `.info()`
  - `.describe()`

```
print(household_poverty.columns)
```

```
Index(['male', 'hh_ID', 'rooms', 'males_tot',
'age', 'ppl_total', 'num_child',
       'bedrooms', 'dependency_rate',
'computer', 'disabled_ppl',
       'water_inside', 'years_of_schooling',
'Target'],
      dtype='object')
```

```
print(household_poverty.dtypes)
```

```
male                    int64
hh_ID                  object
rooms                   int64
males_tot               int64
age                     int64
ppl_total               int64
num_child               int64
bedrooms                int64
dependency_rate         int64
computer                int64
disabled_ppl            int64
water_inside            int64
years_of_schooling      int64
Target                  int64
dtype: object
```

Reviewing household_poverty data - info

```
print(household_poverty.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 14 columns):
male                  1000 non-null int64
hh_ID                 1000 non-null object
rooms                 1000 non-null int64
males_tot             1000 non-null int64
age                   1000 non-null int64
ppl_total             1000 non-null int64
num_child             1000 non-null int64
bedrooms              1000 non-null int64
dependency_rate       1000 non-null int64
computer              1000 non-null int64
disabled_ppl          1000 non-null int64
water_inside          1000 non-null int64
years_of_schooling    1000 non-null int64
Target                1000 non-null int64
dtypes: int64(13), object(1)
memory usage: 109.5+ KB
None
```

Reviewing household_poverty data - describe

```
print(household_poverty.describe())
```

```
              male        rooms  ...  years_of_schooling       Target
count  1000.000000  1000.000000  ...         1000.000000  1000.000000
mean      0.477000     5.244000  ...            8.055000     3.586000
std       0.499721     1.692728  ...            4.898568     0.833846
min       0.000000     1.000000  ...            0.000000     1.000000
25%       0.000000     4.000000  ...            5.000000     4.000000
50%       0.000000     5.000000  ...            8.000000     4.000000
75%       1.000000     6.000000  ...           11.000000     4.000000
max       1.000000    11.000000  ...           21.000000     4.000000

[8 rows x 13 columns]
```

Reviewing household_poverty data - index

- What is the index of this dataframe?

```
print(household_poverty.index)
```

```
RangeIndex(start=0, stop=1000, step=1)
```

- Remember, we can set the index as one of our columns
- What would make most sense with this dataset?

```
household_poverty = household_poverty.set_index('hh_ID')
print(household_poverty.index)
```

```
Index(['21eb7fcc1', '0e5d7a658', '2c7317ea8', '2b58d945f', '2b58d945f',
       '2b58d945f', '2b58d945f', 'd6dae86b7', 'd6dae86b7', 'd6dae86b7',
       ...
       'f54bc4b11', 'f54bc4b11', 'f54bc4b11', 'f54bc4b11', '34bab1f1f',
       '34bab1f1f', '34bab1f1f', '34bab1f1f', '52cf8fe9d'],
      dtype='object', name='hh_ID', length=1000)
```

- Now we can look up rows by the actual household IDs

Looking up by household ID

- Let's refresh looking up by index
- We can use `.loc` to look up by specific household ID

```
# Look up a specific row by index.
print(household_poverty.loc['21eb7fcc1'])
```

```
male                  1
rooms                 3
males_tot             1
age                  43
ppl_total             1
num_child             0
bedrooms              1
dependency_rate      30
computer              0
disabled_ppl          0
water_inside          1
years_of_schooling   10
Target                4
Name: 21eb7fcc1, dtype: int64
```

- **When would something like this be useful in your data?**

Looking up by household ID

- We can use `.iloc` to look up by row number of the index

```
# Look up a specific row by index.
print(household_poverty.iloc[1])
```

```
male                    1
rooms                   4
males_tot               1
age                    67
ppl_total               1
num_child               0
bedrooms                1
dependency_rate        29
computer                0
disabled_ppl            0
water_inside            1
years_of_schooling     12
Target                  4
Name: 0e5d7a658, dtype: int64
```

- And finally, we can reset our index back to the original index

```
household_poverty = household_poverty.reset_index()
```

Knowledge check 3

Exercise 3

Module completion checklist

Objective	Complete
Work with numpy objects	✔
Summarize use cases of pandas and update directory settings	✔
Demonstrate use of basic operations on series	✔
Demonstrate use of basic operations on dataframes	✔
Load data into Python using pandas	✔
Summarize data using pandas	

Methods to summarize and group data in pandas

- What if we want more detailed summary metrics? Use `groupby()`!
- `groupby()` describes a process involving the following steps:
  - **splitting** the data into groups based on some criteria
  - **applying** a function to each group independently
- We'll be starting with the most straightforward part of `groupby()`, the split step

Splitting using groupby()

- A string passed to `groupby()` may refer to either a column or an index level
- We can either group by column or by index
- This fits into the **splitting** step, as we are splitting the data to be grouped by number of rooms
- We will group by the column `rooms` for now

```
grouped = household_poverty.groupby('rooms')
print(grouped.first())
```

```
           hh_ID  male  males_tot  ...  water_inside  years_of_schooling  Target
rooms                              ...
1       3e16fab89     1          2  ...             0                   6       4
2       d6dae86b7     0          1  ...             1                   0       4
3       21eb7fcc1     1          1  ...             1                  10       4
4       0e5d7a658     1          1  ...             1                  12       4
5       2b58d945f     1          2  ...             1                   9       4
6       65d20b573     0          2  ...             1                  14       4
7       bcc196e5a     0          1  ...             1                  13       4
8       2c7317ea8     0          0  ...             1                  11       4
9       6f1edad4e     1          3  ...             1                  13       4
10      bdd842cfd     1          2  ...             1                  11       4
11      b64f4194f     1          1  ...             1                   9       4

[11 rows x 13 columns]
```

Summarizing using groupby()

- All the summary functions can be applied to a group
- For a refresher, here are the summary functions:

Function	Description
count	Number of non-null observations
sum	Number of non-null observations
max	Maximum of values
min	Minimum of values
mean	Mean of values
median	Arithmetic median of values
var	Variance of each object
std	Standard deviation of each object

Groupby() and summary functions

- We can now move to the second step of summarizing data, **applying** a function to the group
- Let's see the distribution of households by room

```
# We are counting the number of hh_IDs by number of rooms, and creating a dataframe.
hh_ID = grouped.count()[['hh_ID']]
print(hh_ID)
```

```
       hh_ID
rooms
1          8
2         19
3         90
4        227
5        291
6        156
7        112
8         59
9         14
10        17
11         7
```

```
# This syntax would do the same, but create a series.
print(grouped.count().hh_ID)
```

```
rooms
1          8
2         19
3         90
4        227
5        291
6        156
7        112
8         59
9         14
10        17
11         7
Name: hh_ID, dtype: int64
```

A little more about summarizing - sorting

- **Sorting**: ordering row(s) by value of column, either low to high (default) or high to low (`ascending = False`)

```
print(hh_ID.sort_values(by = ['hh_ID'], ascending = [False]))
```

```
       hh_ID
rooms
5        291
4        227
6        156
7        112
3         90
8         59
2         19
10        17
9         14
1          8
11         7
```

A little more about summarizing - filtering

- **Filtering**: using filter(s) to subset only certain aspects of your dataset

```
print(hh_ID.query('rooms < 5'))
```

```
        hh_ID
rooms
1           8
2          19
3          90
4         227
```

A little more about summarizing - new columns

- **Create new columns** by creating a series and adding it to a current `dataframe`

```
over100_hh = hh_ID['hh_ID'] > 100
# Add the new column.
hh_ID['over100_hh'] = over100_hh
print(hh_ID.head())
```

```
        hh_ID  over100_hh
rooms
1           8       False
2          19       False
3          90       False
4         227        True
5         291        True
```

Knowledge check 4

Exercise 4

Module completion checklist

Objective	Complete
Work with numpy objects	✔
Summarize use cases of pandas and update directory settings	✔
Demonstrate use of basic operations on series	✔
Demonstrate use of basic operations on dataframes	✔
Load data into Python using pandas	✔
Summarize data using pandas	✔

Workshop!

- Workshops are to be completed in the afternoon either with a dataset for a capstone project or with another dataset of your choosing
- Make sure to **annotate and comment your code** so that it is easy for others to understand what you are doing
- This is an exploratory exercise to **get you comfortable with the content** we discussed today.'

  Today, you will:

  - Load data into a pandas dataframe. You can use `chicago_census.csv`
  - Inspect your dataframe
  - Perform subsets, sorts, and more on your dataframe
  - Summarize your dataframe after grouping it into groups

This completes our module

Congratulations!