CAP 5768 – Intro to Data Science

Lecture 2 – NumPy, Pandas, and Matplotlib

Oge Marques, PhD

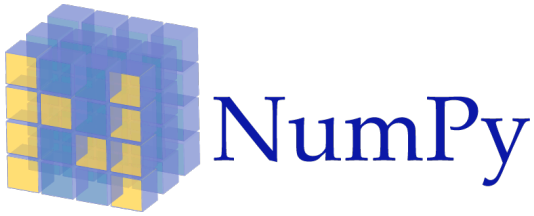
Professor

College of Engineering and Computer Science

College of Business

FAU FLORIDA ATLANTIC UNIVERSITY

NumPy

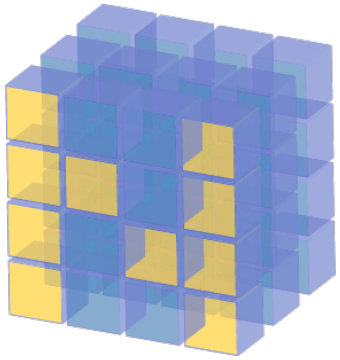


- NumPy (Numerical Python) is “a library for the Python programming language, adding *support for large, multi-dimensional arrays and matrices*, along with a *large collection of high-level mathematical functions* to operate on these arrays.”

- Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data.

- Follow Part 5 of the “Guided Tour” to learn more about NumPy.

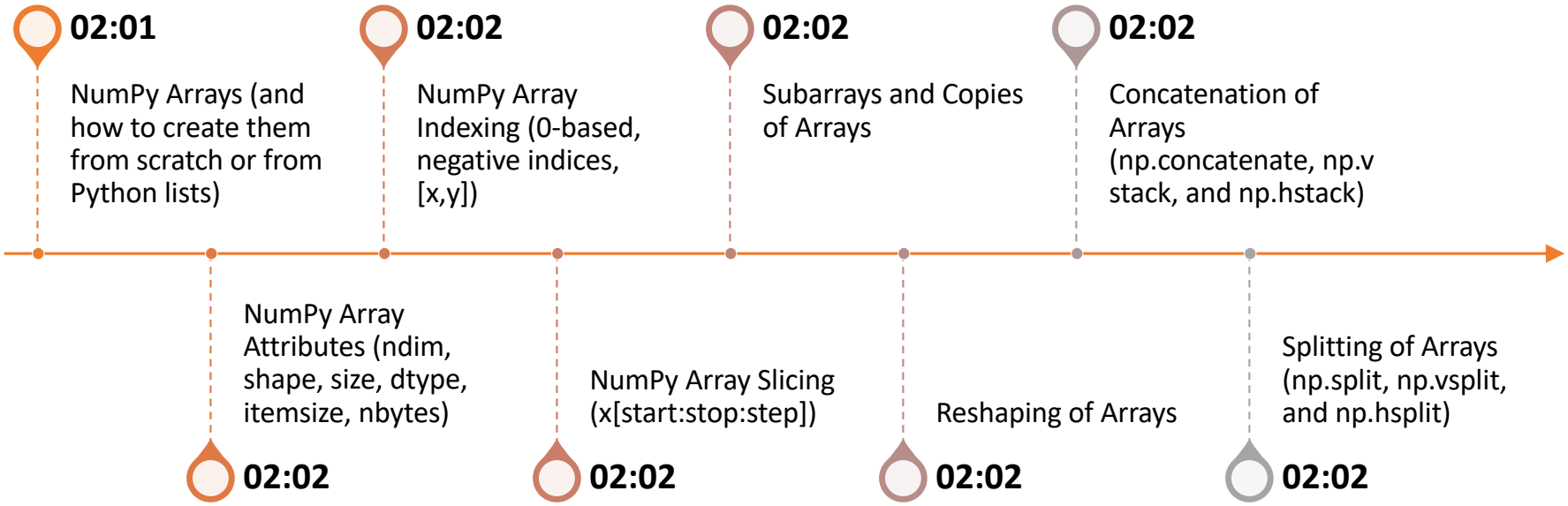
Source: https://en.wikipedia.org/wiki/NumPy and https://www.numpy.org/



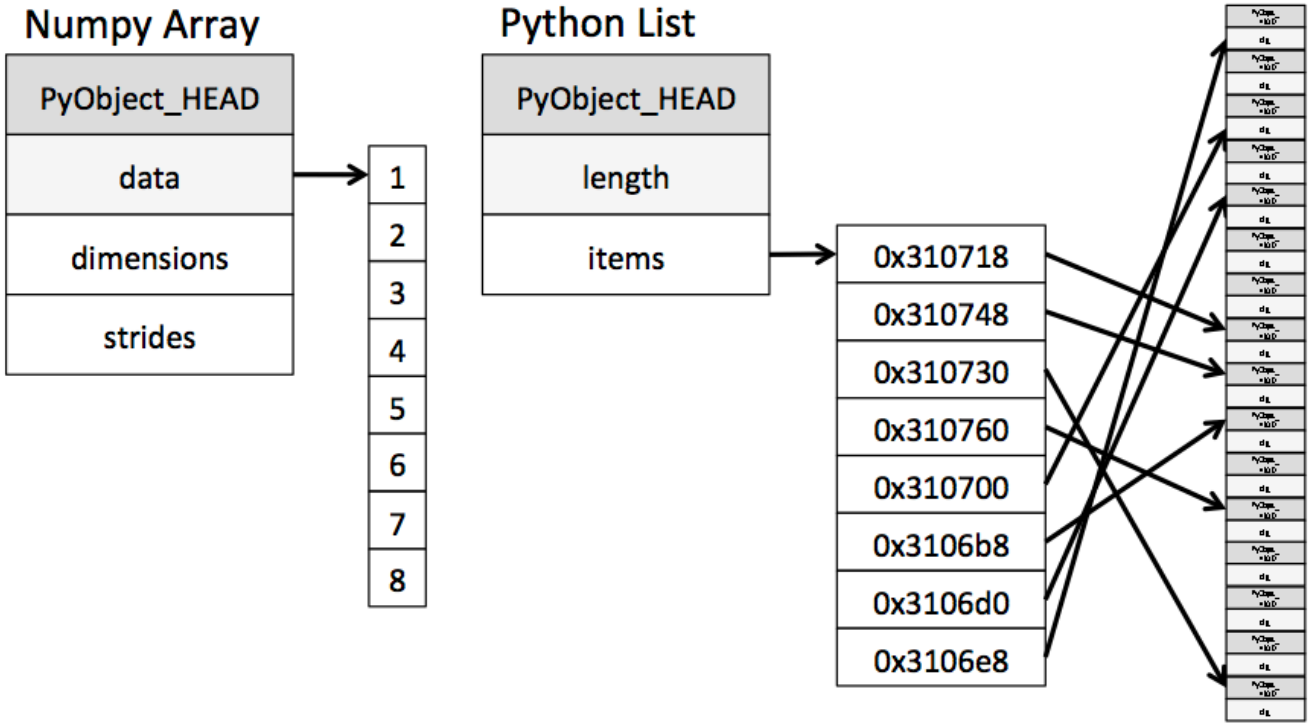
NumPy

- 02.00-Introduction-to-NumPy.ipynb
- 02.01-Understanding-Data-Types.ipynb
- 02.02-The-Basics-Of-NumPy-Arrays.ipynb
- 02.03-Computation-on-arrays-ufuncs.ipynb
- 02.04-Computation-on-arrays-aggregates.ipynb
- 02.05-Computation-on-arrays-broadcasting.ipynb
- 02.06-Boolean-Arrays-and-Masks.ipynb
- 02.07-Fancy-Indexing.ipynb
- 02.08-Sorting.ipynb
- 02.09-Structured-Data-NumPy.ipynb

NumPy: highlights



NumPy arrays vs. Python lists (under the hood)



NumPy 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
`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 Universal Functions (02.03)

- Vectorization, no for loops!
- Array arithmetic (see table)
- Many other operations (applied to array as a whole)
- Aggregates (`reduce` and `accumulate`)
- Outer products

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`)

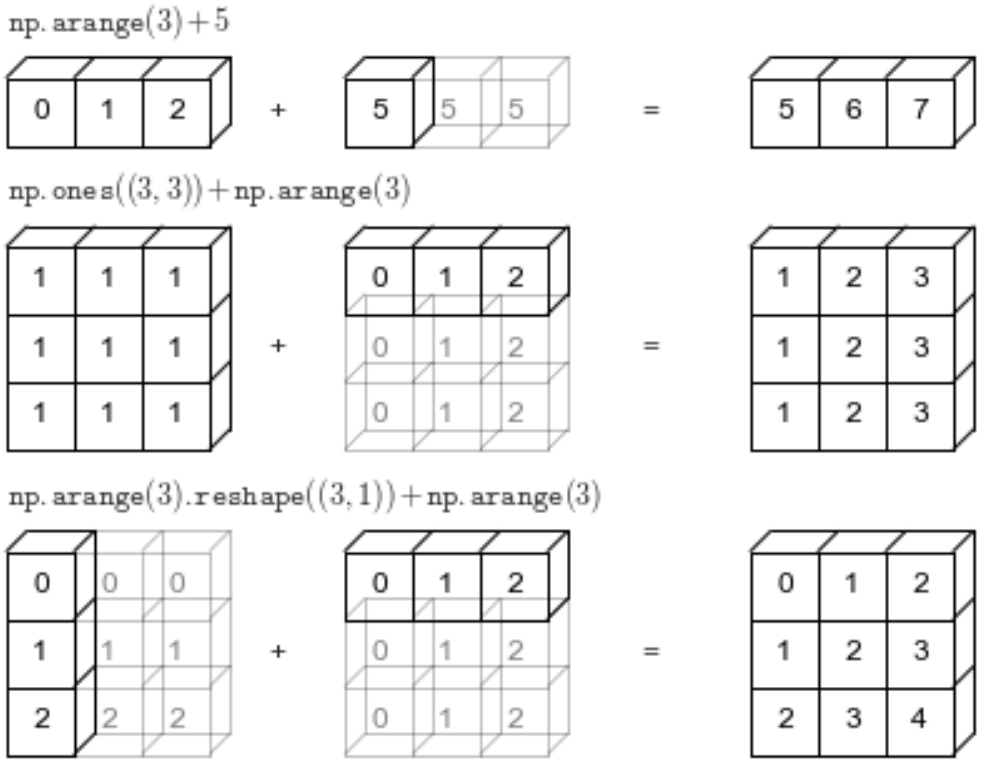
NumPy: aggregations (02.04)

- Beware of the axis parameter!
  - 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.

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
`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

NumPy arrays: broadcasting (02.05)

- Broadcasting allows binary operations to be performed on arrays of different sizes.



NumPy arrays: broadcasting (02.05)

Rules of broadcasting

- 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.

Comparisons, masks, and Boolean logic (02.06)

Comparison operators

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

Comparisons, masks, and Boolean logic (02.06)

Boolean arrays as masks ("logical indexing" in MATLAB)

AND and OR:
keywords (and, or) or
operators (&, |)?

`np.all` **and** `np.any`

NumPy: last bits

02:07

Fancy indexing

- Fancy indexing is indexing with an array of indices.

02:08

Sorting

- Built-in functions: np.sort and np.argsort

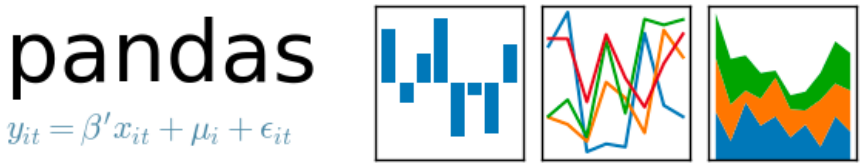
02:09

Structured arrays

- A reminder that we need something better than NumPy here…
  Enter Pandas!


Hands on!

Pandas



- Pandas (Python Data Analysis Library) is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

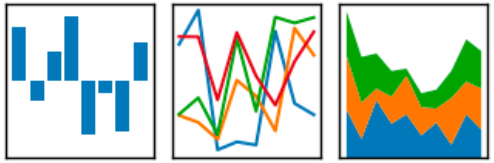
- It offers data structures and operations for manipulating numerical tables and time series.

- Follow Part 6 of the “Guided Tour” to to learn more about Pandas.

Source: https://pandas.pydata.org/ and https://en.wikipedia.org/wiki/Pandas_(software)

pandas

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



- 03.00-Introduction-to-Pandas.ipynb
- 03.01-Introducing-Pandas-Objects.ipynb
- 03.02-Data-Indexing-and-Selection.ipynb
- 03.03-Operations-in-Pandas.ipynb
- 03.04-Missing-Values.ipynb
- 03.05-Hierarchical-Indexing.ipynb
- 03.06-Concat-And-Append.ipynb
- 03.07-Merge-and-Join.ipynb
- 03.08-Aggregation-and-Grouping.ipynb
- 03.09-Pivot-Tables.ipynb
- 03.10-Working-With-Strings.ipynb
- 03.11-Working-with-Time-Series.ipynb
- 03.12-Performance-Eval-and-Query.ipynb
- 03.13-Further-Resources.ipynb

The Pandas Series object (03.01)

- A Pandas Series is a one-dimensional array of indexed data that is much more general and flexible than the one-dimensional NumPy array that it emulates
- 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.
  - Consequently, the index need not be an integer, but can consist of values of any desired type.
- A Pandas Series can be thought of as 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.

The Pandas DataFrame object (03.01)

- A DataFrame is an analog of a 2D array with both flexible row indices and flexible column names.
- Just as you might think of a 2D array as an ordered sequence of aligned 1D columns, you can think of a DataFrame as a sequence of aligned Series objects.
- The DataFrame can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary.

The Pandas Index object (03.01)

- Both the Series and DataFrame objects contain an explicit *index* that lets you reference and modify data.
- This Index object is an interesting structure in itself, and it can be thought of either as an *immutable array* or as an *ordered set* (technically a multi-set, as Index objects may contain repeated values).

Data indexing and selection (03.02)

- Pandas Series and DataFrame objects support many means of accessing and modifying their values, including: indexing, slicing, masking, fancy indexing, and combinations thereof.
- Beware of syntax aspects!

```
In [1]: import pandas as pd
        data = pd.Series([0.25, 0.5, 0.75, 1.0],
                         index=['a', 'b', 'c', 'd'])
        data

Out[1]: a    0.25
        b    0.50
        c    0.75
        d    1.00
        dtype: float64
```

```
In [7]: # slicing by explicit index
        data['a':'c']

Out[7]: a    0.25
        b    0.50
        c    0.75
        dtype: float64
```

```
In [8]: # slicing by implicit integer index
        data[0:2]

Out[8]: a    0.25
        b    0.50
        dtype: float64
```

Indexers: loc, iloc, and ix (03.02)

- 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.
  - Note: ix is deprecated.
- **Attention**! While *indexing* refers to columns, *slicing* refers to rows.

Operating on Data in Pandas (03.03)

Index preservation

Index alignment

NaN (more on this in 03.04)

Operations between a DataFrame and a Series are similar to operations between a 2D and 1D NumPy array.

Operating on Data in Pandas (03.03)

Table 3-1. Mapping between Python operators and Pandas methods

Python operator	Pandas method(s)
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

Handling Missing Data (03.04)

- We'll refer to missing data in general as *null*, *NaN*, or *NA* values.
- 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 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.
  - *NaN* and *None* both have their place, and Pandas is built to handle the two of them nearly interchangeably, converting between them where appropriate.

Handling Missing Data (03.04)

Upcasting conventions in Pandas when NA values are introduced

Typeclass	Conversion When Storing NAs	NA Sentinel Value
`floating`	No change	`np.nan`
`object`	No change	`None` or `np.nan`
`integer`	Cast to `float64`	`np.nan`
`boolean`	Cast to `object`	`None` or `np.nan`

Handling Missing Data (03.04)

Useful methods for detecting, removing, and replacing null values in Pandas data structures

- `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

Handling Missing Data (03.04)

- Pandas data structures have two useful methods for detecting null data: **isnull()** and **notnull()**.
  - Either one will return a Boolean mask over the data.
- Dropping null values
  - There are the convenience methods, **dropna()** (which removes NA values) and **fillna()** (which fills in NA values).
- Warning!
  - 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.
    - The how and thresh parameters allow fine control of the number of nulls to allow through.

Handling Missing Data (03.04)

Table 3-2. Pandas handling of NAs by type

Typeclass	Conversion when storing NAs	NA sentinel value
`floating`	No change	`np.nan`
`object`	No change	None or `np.nan`
`integer`	Cast to `float64`	`np.nan`
`boolean`	Cast to `object`	None or `np.nan`

Hierarchical Indexing (03.05)

- The Pandas MultiIndex type
- Several methods of MultiIndex creation
- Indexing and Slicing a MultiIndex
- Rearranging Multi-Indices
- Data Aggregations on Multi-Indices

Combining Datasets: Concat and Append (03.06)

- pd.concat()
  - Handling duplicate indexes
- Concatenation with joins
- The append() method

Combining Datasets: Merge and Join (03.07)

- pd.merge() -- three types of join: *one-to-one*, *many-to-one*, and *many-to-many*
- Specification of the merge key
  - on
  - left_on
  - right_on
  - left_index
  - right_index
- Specifying Set Arithmetic for Joins
- Overlapping Column Names: The *suffixes* Keyword

Aggregation and Grouping (03.08)

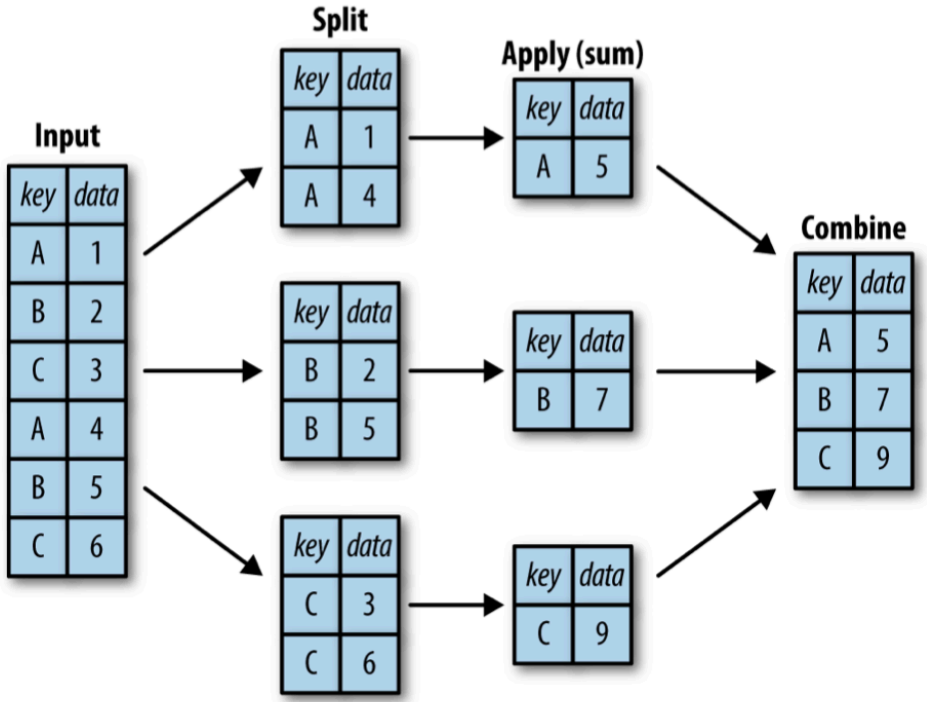
- The describe() method
- Other methods (table 3-3)

Table 3-3. Listing of Pandas aggregation methods

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

Aggregation and Grouping (03.08)

- GroupBy: Split, Apply, Combine
  - The *split* step involves breaking up and grouping a DataFrame depending on the value of the specified key.
  - The *apply* step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.
  - The *combine* step merges the results of these operations into an output array.



Pivot tables (03.09)

The pivot table takes simple column-wise data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data.

The difference between pivot tables and GroupBy can sometimes cause confusion; think of pivot tables as essentially *multidimensional* version of GroupBy aggregation.

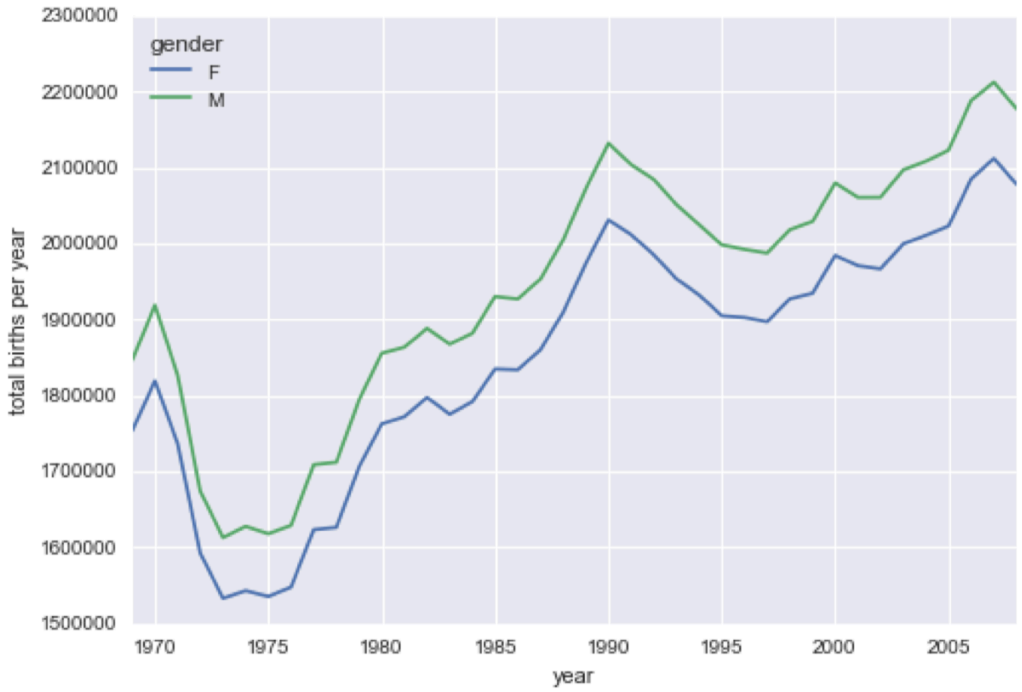
That is, you split-apply-combine, but both the split and the combine happen across not a one-dimensional index, but across a two-dimensional grid.

Pivot tables (03.09)

- Example: Titanic
  - pivot_table
  - cut
  - qcut
  - aggfunc

Pivot tables (03.09)

- Example: Birthrate data



Working with strings (03.10)

- Pandas provides a comprehensive set of *vectorized string operations* that become an essential piece of the type of munging required when working with (i.e., *cleaning up*) *real-world data*.

`len()`	`lower()`	`translate()`	`islower()`
`ljust()`	`upper()`	`startswith()`	`isupper()`
`rjust()`	`find()`	`endswith()`	`isnumeric()`
`center()`	`rfind()`	`isalnum()`	`isdecimal()`
`zfill()`	`index()`	`isalpha()`	`split()`
`strip()`	`rindex()`	`isdigit()`	`rsplit()`
`rstrip()`	`capitalize()`	`isspace()`	`partition()`
`lstrip()`	`swapcase()`	`istitle()`	`rpartition()`

Working with strings (03.10)

Regular expressions

Table 3-4. Mapping between Pandas methods and functions in Python's `re` module

Method	Description
`match()`	Call `re.match()` on each element, returning a Boolean.
`extract()`	Call `re.match()` on each element, returning matched groups as strings.
`findall()`	Call `re.findall()` on each element.
`replace()`	Replace occurrences of pattern with some other string.
`contains()`	Call `re.search()` on each element, returning a Boolean.
`count()`	Count occurrences of pattern.
`split()`	Equivalent to `str.split()`, but accepts regexps.
`rsplit()`	Equivalent to `str.rsplit()`, but accepts regexps.

Working with strings (03.10)

Miscellaneous methods

Table 3-5. Other Pandas string methods

Method	Description
`get()`	Index each element
`slice()`	Slice each element
`slice_replace()`	Replace slice in each element with passed value
`cat()`	Concatenate strings
`repeat()`	Repeat values
`normalize()`	Return Unicode form of string
`pad()`	Add whitespace to left, right, or both sides of strings
`wrap()`	Split long strings into lines with length less than a given width
`join()`	Join strings in each element of the `Series` with passed separator
`get_dummies()`	Extract dummy variables as a `DataFrame`

Working with time series (03.11)

- Pandas was developed in the context of financial modeling, so as you might expect, it contains a fairly extensive set of tools for working with dates, times, and time-indexed data.
- Date and time data comes in a few flavors:
  - *Time stamps* reference particular moments in time (e.g., July 4th, 2015 at 7:00am).
  - *Time intervals* and *periods* reference a length of time between a particular beginning and end point; for example, the year 2015. Periods usually reference a special case of time intervals in which each interval is of uniform length and does not overlap (e.g., 24 hour-long periods comprising days).
  - *Time deltas* or *durations* reference an exact length of time (e.g., a duration of 22.56 seconds).

Working with time series (03.11)

- Pandas provides a **Timestamp** object, which combines the ease-of-use of **datetime** and **dateutil** with the efficient storage and vectorized interface of **numpy.datetime64**.
- From a group of these Timestamp objects, Pandas can construct a *DatetimeIndex* that can be used to index data in a *Series* or *DataFrame.*
- See examples in the book.

Working with time series (03.11)

- For *time stamps*, Pandas provides the **Timestamp** type.
  - The associated Index structure is DatetimeIndex.
- For *time Periods*, Pandas provides the **Period** type.
  - The associated index structure is PeriodIndex.
- For *time deltas* or *durations*, Pandas provides the **Timedelta** type.
  - The associated index structure is TimedeltaIndex.

Working with time series (03.11)

- Date ranges
  - To make the creation of regular date sequences more convenient, Pandas offers a few functions for this purpose:
    - pd.date_range() for timestamps
    - pd.period_range() for periods
    - pd.timedelta_range() for time deltas.
  - We've seen that Python's range() and NumPy's np.arange() turn a startpoint, endpoint, and optional stepsize into a sequence.
    - Similarly, pd.date_range() accepts a start date, an end date, and an optional frequency code to create a regular sequence of dates. By default, the frequency is one day.

Working with time series (03.11)

- Frequencies and offsets

Table 3-8. Listing of start-indexed frequency codes

Code	Description
MS	Month start
BMS	Business month start
QS	Quarter start
BQS	Business quarter start
AS	Year start
BAS	Business year start

Table 3-7. Listing of Pandas frequency codes

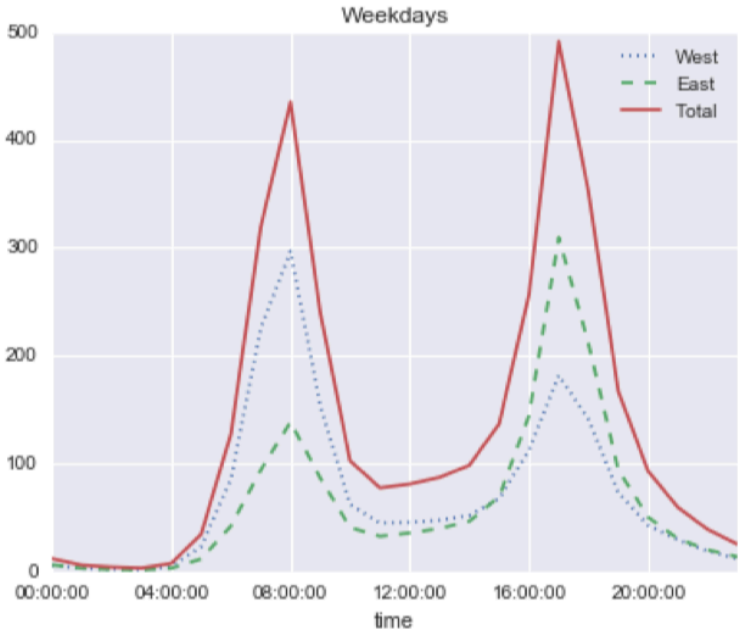
Code	Description	Code	Description
D	Calendar day	B	Business day
W	Weekly		
M	Month end	BM	Business month end
Q	Quarter end	BQ	Business quarter end
A	Year end	BA	Business year end
H	Hours	BH	Business hours
T	Minutes		
S	Seconds		
L	Milliseonds		
U	Microseconds		
N	Nanoseconds		

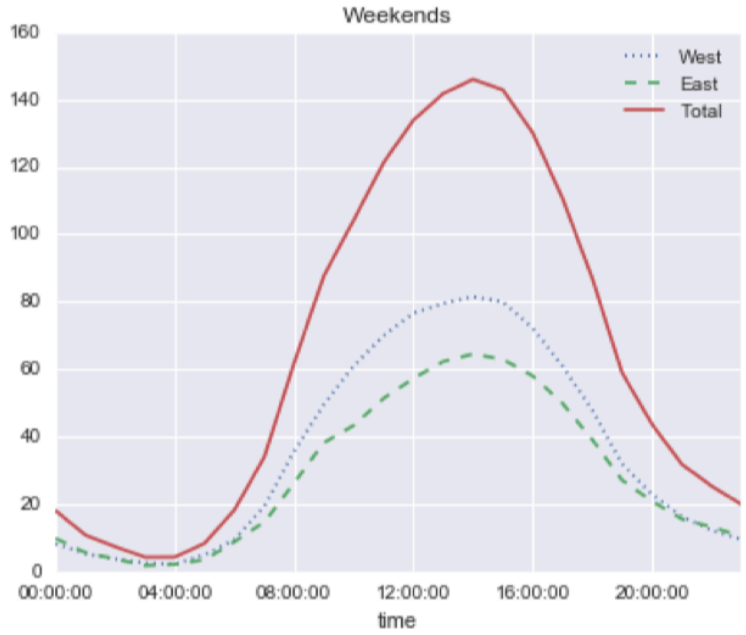
Working with time series (03.11)

- Resampling
- Shifting
- Windowing

Working with time series (03.11)

Example: visualizing Seattle bicycle counts





High-Performance Pandas: eval() and query() (03.12)

(OPTIONAL)


Hands on!

Matplotlib

- Matplotlib is a 2D **plotting library** for the Python programming language and its numerical mathematics extension NumPy.
- Matplotlib produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.
- Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and several GUI toolkits.

- Follow Part 7 of the “Guided Tour” to to learn more about Matplotlib.

Source: https://en.wikipedia.org/wiki/Matplotlib and https://matplotlib.org/

- 04.00-Introduction-To-Matplotlib.ipynb
- 04.01-Simple-Line-Plots.ipynb
- 04.02-Simple-Scatter-Plots.ipynb
- 04.03-Errorbars.ipynb
- 04.04-Density-and-Contour-Plots.ipynb
- 04.05-Histograms-and-Binnings.ipynb
- 04.06-Customizing-Legends.ipynb
- 04.07-Customizing-Colorbars.ipynb
- 04.08-Multiple-Subplots.ipynb
- 04.09-Text-and-Annotation.ipynb
- 04.10-Customizing-Ticks.ipynb
- 04.11-Settings-and-Stylesheets.ipynb
- 04.12-Three-Dimensional-Plotting.ipynb
- 04.13-Geographic-Data-With-Basemap.ipynb
- 04.14-Visualization-With-Seaborn.ipynb
- 04.15-Further-Resources.ipynb

Matplotlib: background

- Matplotlib is a multi-platform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack.
- 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.
- Matplotlib works 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.
- There have been new packages that build on its powerful internals to drive Matplotlib via cleaner, more modern APIs—for example, Seaborn.

Matplotlib basics

- Import
- Show
- Saving figures to file
- Two interfaces
  - MATLAB-style
  - Object-oriented

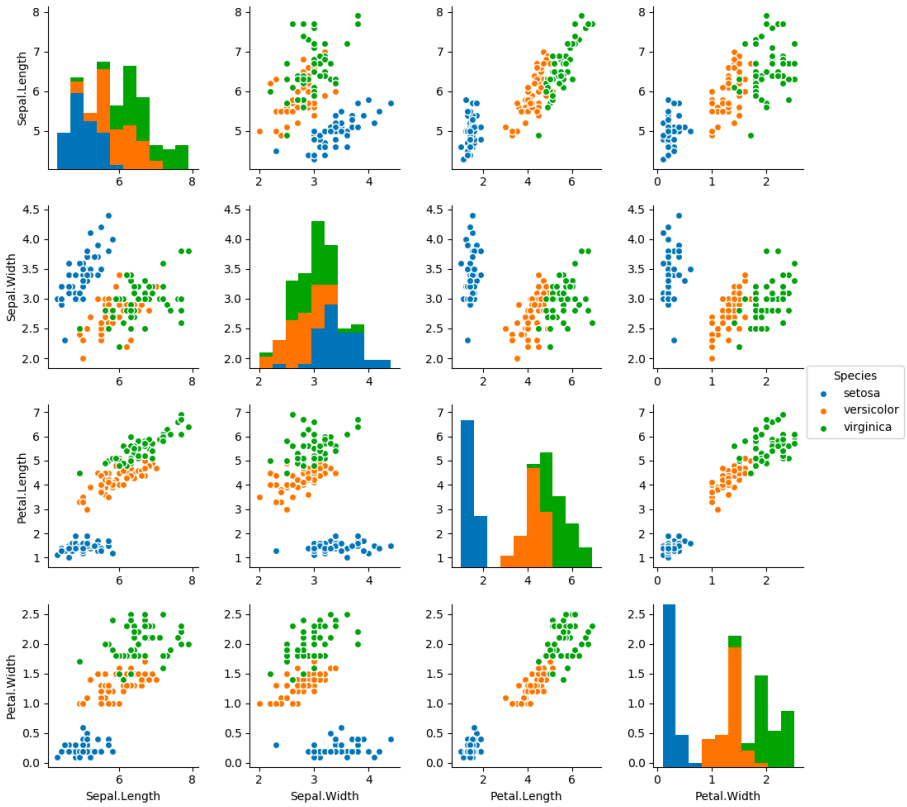
```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

```
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));
```

```
# 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));
```

Matplotlib

- Line plots
- Scatter plots
- Histograms
- Error bars
- Density and contour plots
- Legends, color bars, ticks, text, annotation, and customization options
- Multiple subplots
- Configurations and stylesheets



Seaborn

Seaborn provides an API on top of Matplotlib that offers sane choices for plot style and color defaults, defines simple high-level functions for common statistical plot types, and integrates with the functionality provided by Pandas DataFrames.

Example: Exploring Marathon Finishing Times


Hands on!

Assignment 1: The Python data science stack

Goals:

- To get acquainted with Python and Jupyter notebooks.
- To acquire a basic understanding of the Python "data science stack" (NumPy, Pandas, Matplotlib).
- To have an early experience of manipulating, summarizing, and visualizing small datasets.
- To demonstrate the ability to write Python code to answer questions and test hypotheses based on the contents of those datasets.

Assignment 1: The Python data science stack


Hands on!

Assignment 2: Exploratory Data Analysis

Goals:

- To increase familiarity with the Python "data science stack" (NumPy, Pandas, Matplotlib).
- To explore (manipulate, summarize, and visualize) datasets.
- To improve the ability to write Python code to answer questions and test hypotheses based on the contents of those datasets.

Assignment 2: Exploratory Data Analysis

- MovieLens 1M
- Titanic
- US Baby Names (1880-2018)


Hands on!