4. Working with structured data

Athina Tzovara

University of Bern

Athina.Tzovara@unibe.ch

Last week

1. Numerical data types

2. Comparisons; masks and Boolean logic

3. Broadcasting

Today

1. Multimodal data & Pandas
2. Basic data structures: Series and DataFrames
3. Selecting data in Series and DataFrames
4. Ufuncs in data frames

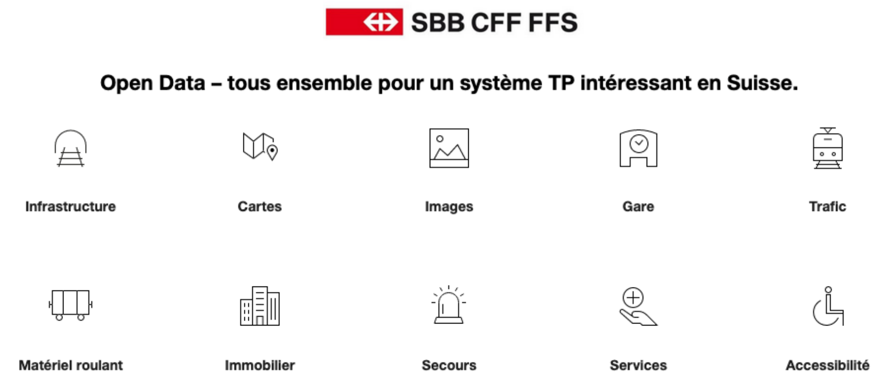
Why do we need multimodal data?

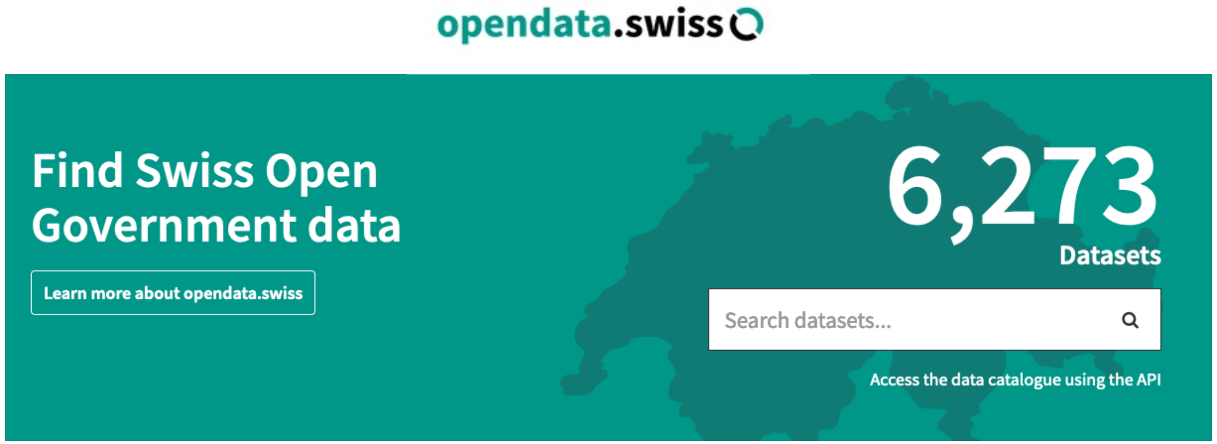
Most datasets around us contain multiple data types

Examples?

Why do we need multimodal data?

Most datasets around us contain multiple data types

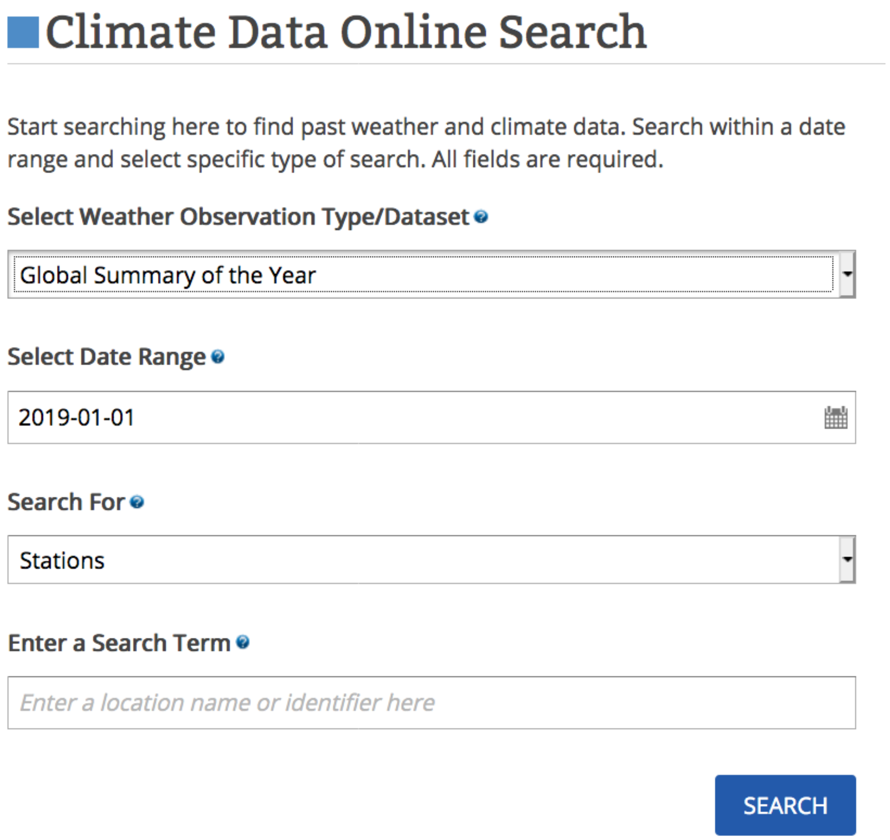




23andMe®

CSV files


NOAA NATIONAL CENTERS FOR ENVIRONMENTAL INFORMATION
NATIONAL OCEANIC AND ATMOSPHERIC ADMINISTRATION
Home
Climate Information
Data Access
Customer Support
Contact
About
Search
Home > Climate Data Online
Datasets | Search Tool | Mapping Tool | Data Tools | Help
Climate Data Online
Climate Data Online (CDO) provides free access to NCDC's archive of global historical weather and climate data in addition to station history information. These data include quality controlled daily, monthly, seasonal, and yearly measurements of temperature, precipitation, wind, and degree days as well as radar data and 30-year Climate Normals. Customers can also order most of these data as certified hard copies for legal use.


Climate Data Online Search
Start searching here to find past weather and climate data. Search within a date range and select specific type of search. All fields are required.
Select Weather Observation Type/Dataset
Global Summary of the Year
Select Date Range
2019-01-01
Search For
Stations
Enter a Search Term
Enter a location name or identifier here
SEARCH

CSV files

Data can be easily rendered as a table

	NAME	ELEVATION	DATE	TMAX	TMIN	TAVG
208	GENEVE COINTRIN, SZ	420	1962	14.1	3.9	9.0
209	GENEVE COINTRIN, SZ	420	1963	13.0	4.3	8.6
210	GENEVE COINTRIN, SZ	420	1964	14.2	5.6	9.9
211	GENEVE COINTRIN, SZ	420	1965	13.4	4.5	9.0
212	GENEVE COINTRIN, SZ	420	1966	14.8	5.5	10.2
213	GENEVE COINTRIN, SZ	420	1967	14.6	4.9	9.7
214	GENEVE COINTRIN, SZ	420	1968	13.7	4.7	9.2
215	GENEVE COINTRIN, SZ	420	1969	13.7	4.5	9.1
216	GENEVE COINTRIN, SZ	420	1970	13.9	4.6	9.3
217	GENEVE COINTRIN, SZ	420	1971	14.3	4.4	9.4
218	GENEVE COINTRIN, SZ	420	1972	14.0	4.7	9.4
219	GENEVE COINTRIN, SZ	420	1973	13.9	4.6	9.3
220	GENEVE COINTRIN, SZ	420	1974	14.6	5.2	9.9

Data source: NOAA NATIONAL CENTERS FOR ENVIRONMENTAL INFORMATION NATIONAL OCEANIC AND ATMOSPHERIC ADMINISTRATION

CSV files

Text files where entries are separated by commas

Examples.csv

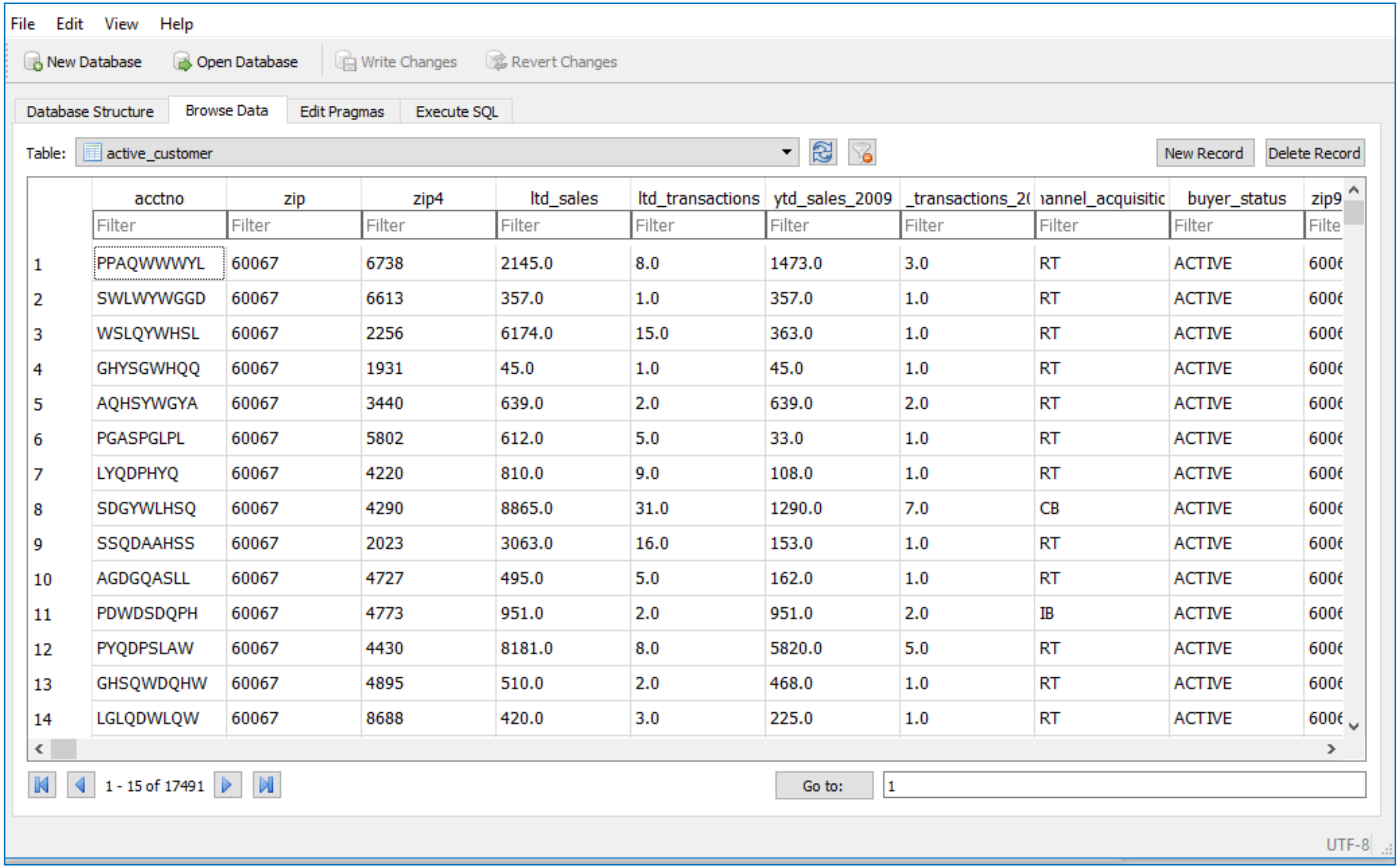
```
STATION,NAME,LATITUDE,LONGITUDE,ELEVATION,DATE,TAVG,TMAX,TMIN
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1962,9,14.1,3.9
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1963,8.6,13,4.3
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1964,9.9,14.2,5.6
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1965,9,13.4,4.5
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1966,10.2,14.8,5.5
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1967,9.7,14.6,4.9
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1968,9.2,13.7,4.7
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1969,9.1,13.7,4.5
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1970,9.3,13.9,4.6
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1971,9.4,14.3,4.4
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1972,9.4,14,4.7
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1973,9.3,13.9,4.6
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1974,9.9,14.6,5.2
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1975,9.8,14.2,5.4
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1976,9.9,14.7,5
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1977,10.1,14.4,5.9
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1978,9.2,13.6,4.8
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1979,9.9,14.4,5.4
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1980,9.2,13.6,4.8
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1981,9.6,14.3,4.8
SZ000008440,"GENEVE COINTRIN, SZ",46.25,6.1331,420,1982,10.4,15.1,5.8
```

They can be easily rendered as a table

	NAME	ELEVATION	DATE	TMAX	TMIN	TAVG
208	GENEVE COINTRIN, SZ	420	1962	14.1	3.9	9.0
209	GENEVE COINTRIN, SZ	420	1963	13.0	4.3	8.6
210	GENEVE COINTRIN, SZ	420	1964	14.2	5.6	9.9
211	GENEVE COINTRIN, SZ	420	1965	13.4	4.5	9.0
212	GENEVE COINTRIN, SZ	420	1966	14.8	5.5	10.2
213	GENEVE COINTRIN, SZ	420	1967	14.6	4.9	9.7
214	GENEVE COINTRIN, SZ	420	1968	13.7	4.7	9.2
215	GENEVE COINTRIN, SZ	420	1969	13.7	4.5	9.1
216	GENEVE COINTRIN, SZ	420	1970	13.9	4.6	9.3
217	GENEVE COINTRIN, SZ	420	1971	14.3	4.4	9.4
218	GENEVE COINTRIN, SZ	420	1972	14.0	4.7	9.4
219	GENEVE COINTRIN, SZ	420	1973	13.9	4.6	9.3
220	GENEVE COINTRIN, SZ	420	1974	14.6	5.2	9.9

Data source: NOAA NATIONAL CENTERS FOR ENVIRONMENTAL INFORMATION
NATIONAL OCEANIC AND ATMOSPHERIC ADMINISTRATION

SQL example

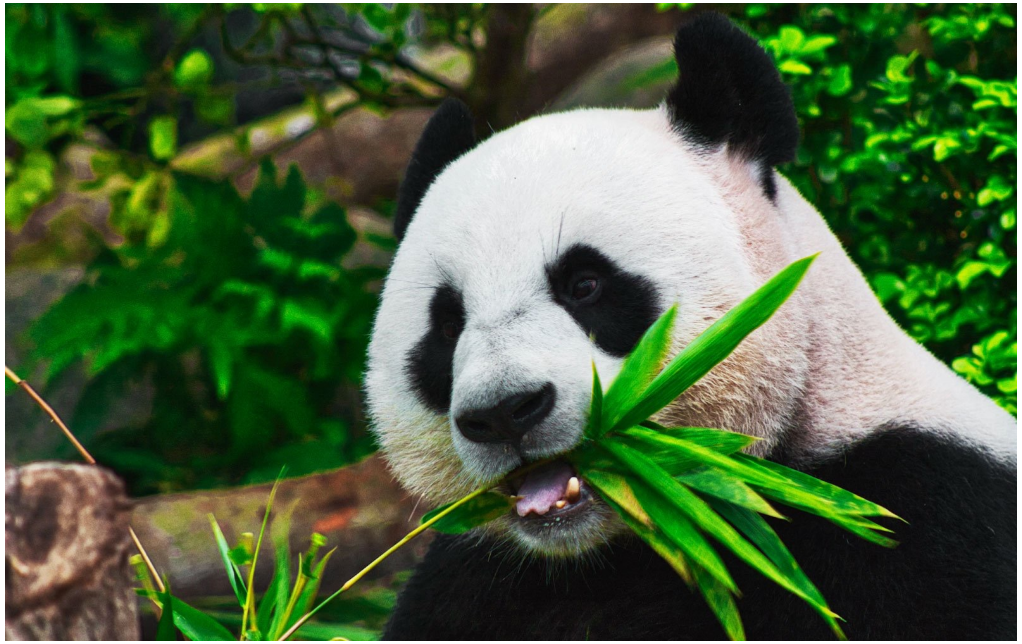


	acctno	zip	zip4	ltd_sales	ltd_transactions	ytd_sales_2009	_transactions_2(	ıannel_acquisitic	buyer_status	zip9
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filte
1	PPAQWWWYL	60067	6738	2145.0	8.0	1473.0	3.0	RT	ACTIVE	6006
2	SWLWYWGGD	60067	6613	357.0	1.0	357.0	1.0	RT	ACTIVE	6006
3	WSLQYWHSL	60067	2256	6174.0	15.0	363.0	1.0	RT	ACTIVE	6006
4	GHYSGWHQQ	60067	1931	45.0	1.0	45.0	1.0	RT	ACTIVE	6006
5	AQHSYWGYA	60067	3440	639.0	2.0	639.0	2.0	RT	ACTIVE	6006
6	PGASPGLPL	60067	5802	612.0	5.0	33.0	1.0	RT	ACTIVE	6006
7	LYQDPHYQ	60067	4220	810.0	9.0	108.0	1.0	RT	ACTIVE	6006
8	SDGYWLHSQ	60067	4290	8865.0	31.0	1290.0	7.0	CB	ACTIVE	6006
9	SSQDAAHSS	60067	2023	3063.0	16.0	153.0	1.0	RT	ACTIVE	6006
10	AGDGQASLL	60067	4727	495.0	5.0	162.0	1.0	RT	ACTIVE	6006
11	PDWDSDQPH	60067	4773	951.0	2.0	951.0	2.0	IB	ACTIVE	6006
12	PYQDPSLAW	60067	4430	8181.0	8.0	5820.0	5.0	RT	ACTIVE	6006
13	GHSQWDQHW	60067	4895	510.0	2.0	468.0	1.0	RT	ACTIVE	6006
14	LGLQDWLQW	60067	8688	420.0	3.0	225.0	1.0	RT	ACTIVE	6006

What types of data files may we encounter?

- Comma Separated Value (CSV)
- JavaScript Object Notation (JSON)
- Structured Query Language (SQL)
- And more!

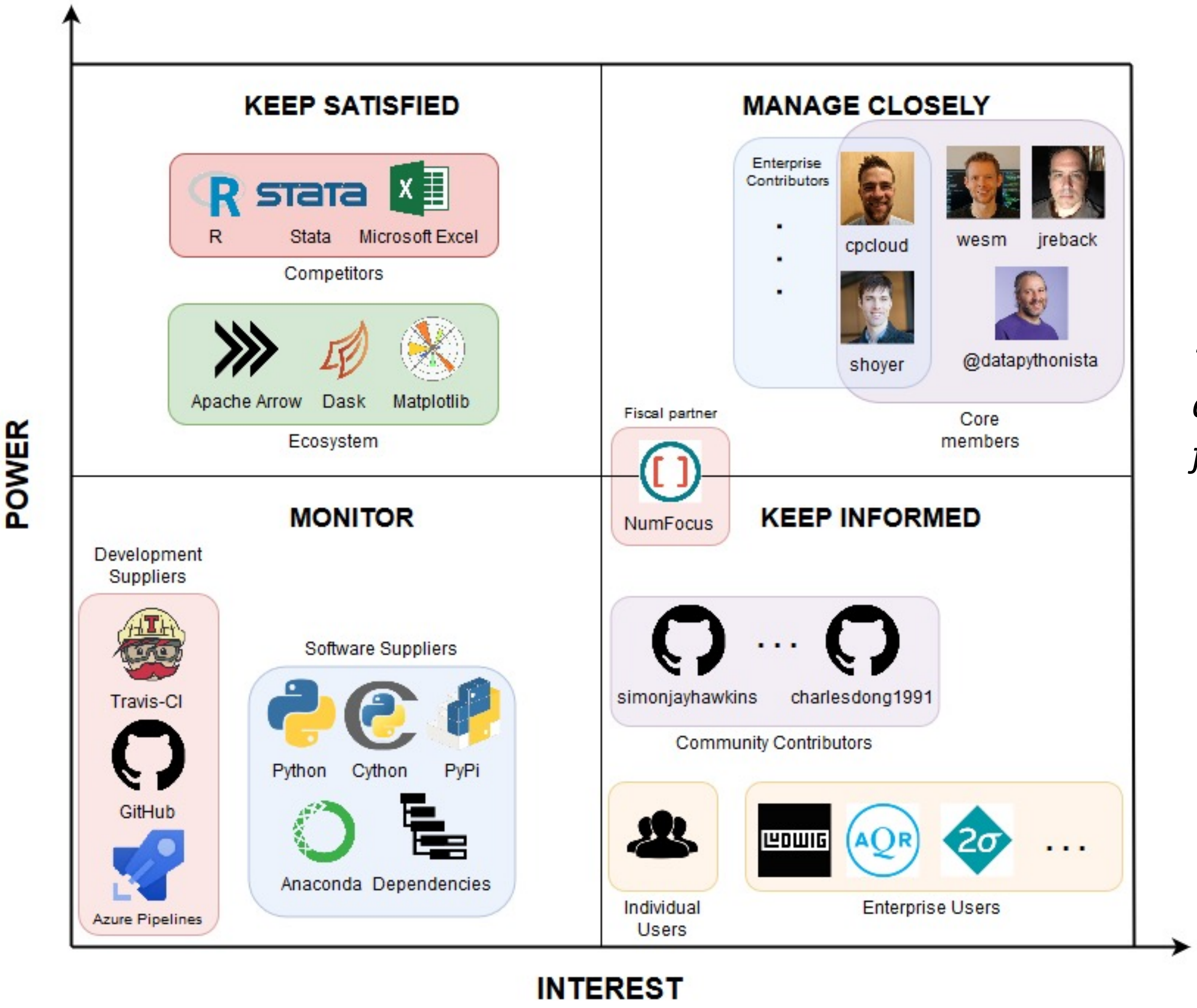
Working with data in python: PANDAS

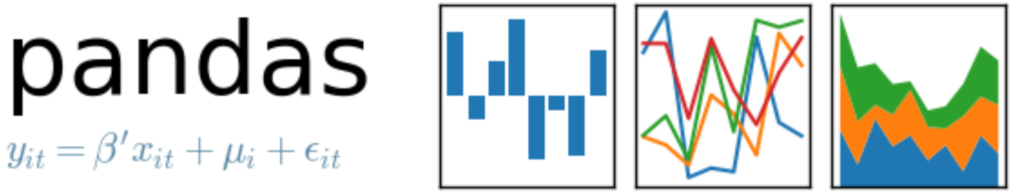


pandas

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

Working with data in python: PANDAS

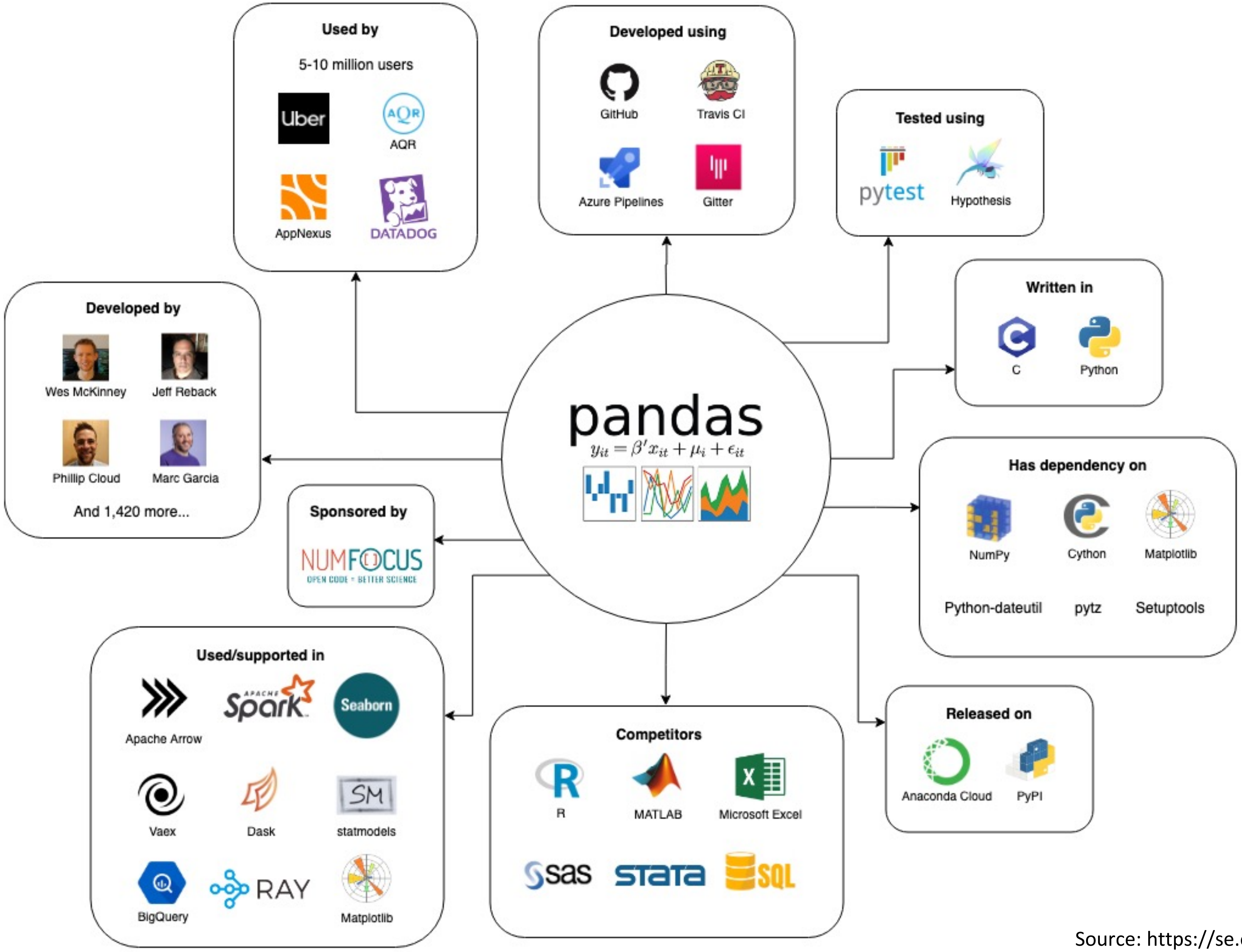




"… the package is meant to accelerate the data analysis workflow and reduce the need for people to deal with technical issues.."

Source: https://se.ewi.tudelft.nl/desosa2019/chapters/pandas/

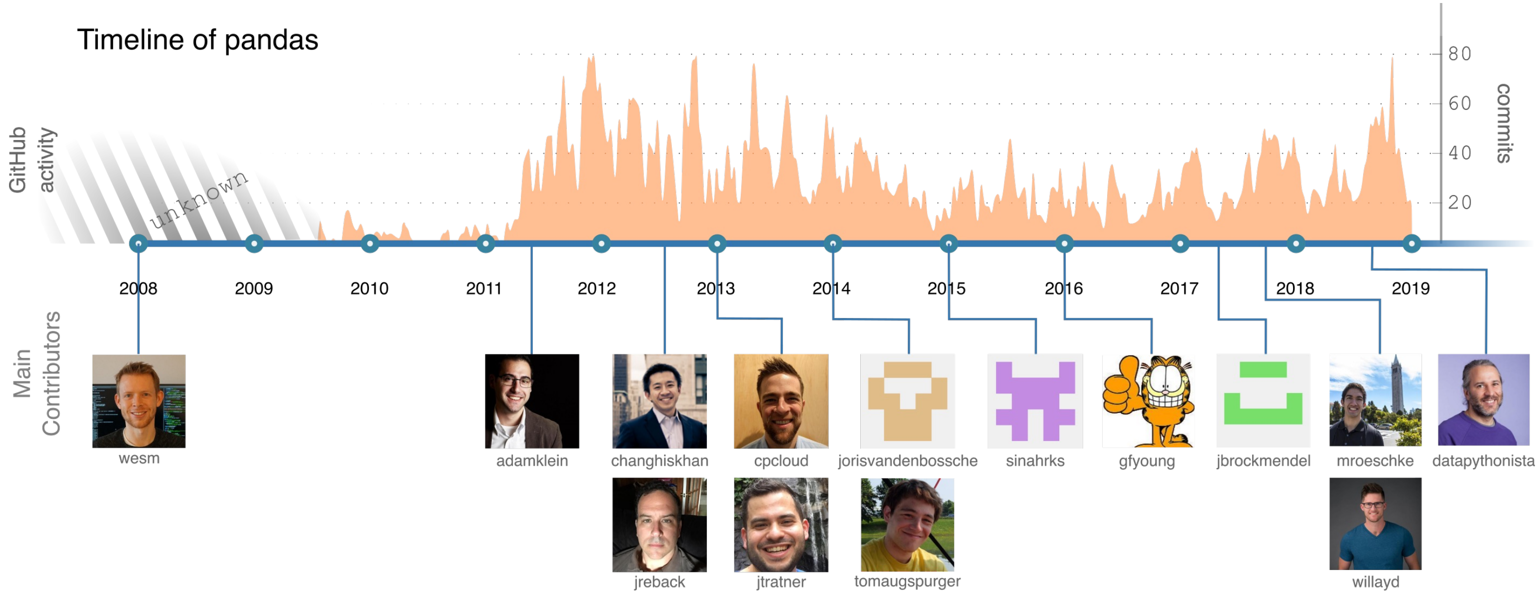
Working with data in python: PANDAS

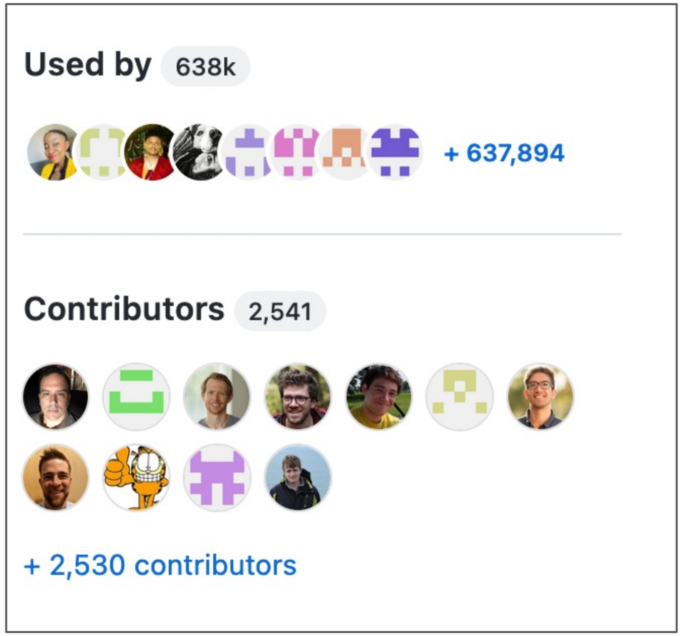


Source: https://se.ewi.tudelft.nl/desosa2019/chapters/pandas/

Working with data in python: PANDAS

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





Users and contributors on 16.03.2022

https://github.com/pandas-dev/pandas

Source: https://se.ewi.tudelft.nl/desosa2019/chapters/pandas/

Working with data in python: PANDAS

pandas

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

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

Introducing Pandas Objects

Enhanced versions of NumPy arrays

Rows and columns are identified with labels

Main structures:

- **Series**
- **DataFrames**

The Pandas Series Object

- 1D array of *indexed* data
- It can be created from a list or array:

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

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

What is different from an array or list?

The Pandas Series Object

- Contains a sequence of **values** and a sequence of **indices**

Values: numpy array

```
data.values
```

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

Index: array-like object of type pd.Index

```
data.index
```

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

The Pandas Series Object

- Data can be accessed by the associated index, like in NumPy arrays:

```
data[1]
```

```
0.5
```

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

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

Series vs. NumPy array

- NumPy Arrays: implicitly defined integer indexes
- **Pandas Series: explicitly defined index associated with values**
- **Pandas Series: index does not need to be integer**

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

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

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

```
0.5
```

Series vs. NumPy array

- NumPy Arrays: implicitly defined integer indexes
- Pandas Series: explicitly defined index associated with values
- Pandas Series: index does not need to be integer, **contiguous, or sequential:**

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=[2, 5, 3, 7])
data
```

```
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

```
data[5]
```

```
0.5
```

Series: as a specialized dictionary

Reminder: Dictionaries

Unordered collections of **unique** values stored in key-value pairs

Mutable: we can modify after their creation

dict = {‘key1’: 10, ‘key2’: 11, ‘key3’: 12}

dict[‘key1’]   dict[‘key2’]   dict[‘key3’]

Series: as a dictionary

- **We can construct a Series object directly from a Python dictionary:**

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

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

Series: as a dictionary

- **We can construct a Series object directly from a Python dictionary:**

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

population
```

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

In older version of Pandas indexes were being ordered.

In more recent versions, we can sort the indexes of our Series with `sort_index()`

Series: as a dictionary

- **We can construct a Series object directly from a Python dictionary:**

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

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

The index for the Series object is drawn from the keys

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

```
38332521
```

We can perform dictionary-like item access

Series: a specialized Python dictionary

- Unlike dictionaries, with Series we can have array-like operations, like slicing:

```
population['Texas':'New York']
```

```
Texas       26448193
New York    19651127
dtype: int64
```

Differences between Series and Dictionaries

Dictionaries

- one of python's default data structures
- Store `key:value` pairs

Series

- one dimensional ndarrays with axis-labels
- Store `array-like; dictionary`
- Built-in structure of Pandas/Numpy
- Efficient: It can use type-specific compiled code that NumPy arrays use

Which to use?

If you "just" need to store `key:value` pairs → Dictionaries may be sufficient

Series has more functionalities and can rely on compiled NumPy type-specific functions

How do we construct a series object?

- Index is an **optional** argument

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

- If left empty, index will be an integer **sequence**

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

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

How do we construct a series object?

- “**data**” can be a scalar, that will be repeated to fill the specified index

- Index can be defined by the user:

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

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

How do we construct a series object?

- "**data**" can also be a dictionary

- Index defaults to dictionary keys:

```
pd.Series({2:'a', 1:'b', 3:'c'})
```

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

- Index can also be set explicitly if a different result is preferred:

```
pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
```

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

Series: Summary

- **Series**:

Equivalent to 1D arrays in NumPy

Key:Value pairs (similar to dictionaries)

Can profit from compiled code

- Now: **DataFrames.**

DataFrames: Equivalent of tables

Series: suitable for 1D data
What happens when we have more variables?

	NAME	ELEVATION	DATE	TMAX	TMIN	TAVG
208	GENEVE COINTRIN, SZ	420	1962	14.1	3.9	9.0
209	GENEVE COINTRIN, SZ	420	1963	13.0	4.3	8.6
210	GENEVE COINTRIN, SZ	420	1964	14.2	5.6	9.9
211	GENEVE COINTRIN, SZ	420	1965	13.4	4.5	9.0
212	GENEVE COINTRIN, SZ	420	1966	14.8	5.5	10.2
213	GENEVE COINTRIN, SZ	420	1967	14.6	4.9	9.7
214	GENEVE COINTRIN, SZ	420	1968	13.7	4.7	9.2
215	GENEVE COINTRIN, SZ	420	1969	13.7	4.5	9.1
216	GENEVE COINTRIN, SZ	420	1970	13.9	4.6	9.3
217	GENEVE COINTRIN, SZ	420	1971	14.3	4.4	9.4
218	GENEVE COINTRIN, SZ	420	1972	14.0	4.7	9.4
219	GENEVE COINTRIN, SZ	420	1973	13.9	4.6	9.3
220	GENEVE COINTRIN, SZ	420	1974	14.6	5.2	9.9

What differences do you observe with respect to 2D NumPy arrays?

Ceating a DataFrame

We can create a DataFrame by combining multiple series objects

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

Dictionary

Series

Series

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

Index attribute

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

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

DataFrames : Equivalent of tables

Like the Series object, `DataFrames` have an **`index`** attribute that gives access to index labels:

```
states.index
```

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

Additionally, they have a **`columns`** attribute which is an `Index` object with column labels:

```
states.columns
```

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

DataFrames: Equivalent to specialized dictionary

`Dictionary`: maps a key to a value
`DataFrame`: maps a column name to a `Series` of column data

E.g. if we access the ‘area’ attribute, a `Series` object will be returned:

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

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

How to construct DataFrames?

A. From a single `Series`

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

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

How to construct DataFrames?

A. From a single Series

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

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

B. From a list of Dictionaries

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

Reminder: list comprehension

	a	b
0	0	0
1	1	2
2	2	4

What if some data are missing?

If some keys are missing from a dictionary, they will be filled in with Nans:

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

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

‘c’ is missing

‘a’ is missing

How to construct DataFrames?

C. From a 2D NumPy array

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

	foo	bar
a	0.865257	0.213169
b	0.442759	0.108267
c	0.047110	0.905718

We specify the column and index names, and we can create a DataFrame from a 2D NumPy array

DataFrame attributes

Python objects have *attributes* and *methods*.

df.attribute	**description**
dtypes	list the types of the columns
columns	list the column names
axes	list the row labels and column names
ndim	number of dimensions
size	number of elements
shape	return a tuple representing the dimensionality
values	numpy representation of the data

DataFrame methods

Python objects have *attributes* and *methods*.

df.method()	description
head([n]), tail([n])	first/last n rows
describe()	generate descriptive statistics (for numeric columns only)
max(), min()	return max/min values for all numeric columns
mean(), median()	return mean/median values for all numeric columns
std()	standard deviation
sample([n])	returns a random sample of the data frame
dropna()	drop all the records with missing values

Today

1. Multimodal data & Pandas
2. Basic data structures: Series and DataFrames
3. **Selecting data in Series and DataFrames**
4. Ufuncs in data frames

Selecting data in Series

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

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

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

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

```
0.5
```

Or, we can use python expressions:

```
'a' in data
```

```
True
```

```
data.keys()
```

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

```
list(data.items())
```

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

Modifying data in Series

We can modify Series objects with a dictionary-like syntax
i.e. we can extend a Series object:

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

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

Series: Mutable Object
Pandas will make decisions about memory layout and data copying

Series as 1D array

Series builds on a dictionary-like interface

Array-style item selection with NumPy array-like mechanisms :

slicing; **masking**; **fancy indexing**

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

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

By explicit index

Slicing examples (observe the output)

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

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

By implicit index

Series as 1D array

Series builds on a dictionary-like interface

Array-style item selection with NumPy array-like mechanisms :

slicing; **masking**; **fancy indexing**

```
# masking
data[(data > 0.3) & (data < 0.8)]
```

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

Masking example

Series as 1D array

Series builds on a dictionary-like interface

Array-style item selection with NumPy array-like mechanisms :

slicing; **masking**; **fancy indexing**

```
# fancy indexing
data[['a', 'e']]
```

```
a    0.25
e    1.25
dtype: float64
```

Fancy Indexing example

Selecting data in DataFrames

DataFrame: as a dictionary:

```
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
```

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

Selecting data in DataFrames

DataFrame: as a dictionary:

```
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
```

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

We can access the individual Series that make up its columns:

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

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

Selecting data in DataFrames

DataFrame: as a dictionary:

```
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
```

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

We can access the individual Series that make up its columns:

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

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

Or, via attribute-style access with column names (if they are strings):

```
data.area
```

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

Selecting data in DataFrames

Method 1: Subset the data frame using column name:

df[‘column_name']

Method 2: Use the column name as an attribute:

df.column_name

Caution! Do not name columns via methods

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

Caution! Do not name columns via methods

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

```python
data.area is data['area']
```

```
True
```

Caution! Do not name columns via methods

Same name for this column as the DataFrame ”pop()” method

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

```
data.area is data['area']
```

```
True
```

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

```
False
```

Modifying DataFrames

We can use the dictionary-like syntax to modify an object, e.g. to add a new column:

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

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

DataFrames vs. NumPy 2D arrays?

Many similarities between Pandas DataFrames and Numpy 2D arrays!
For example:

```
data.values
```

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

```
data.T
```

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

DataFrames vs. NumPy 2D arrays?

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

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

A single index accesses a row

vs.

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

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

A single “index” accesses a column

Bringing DataFrames closer to 2D arrays: loc; iloc; ix

.iloc lets us index the underlying DataFrame array as if it was a NumPy array
DataFrame index and column labels are maintained

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

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127

Bringing DataFrames closer to 2D arrays: loc; iloc; ix

.iloc lets us index the underlying DataFrame array as if it was a NumPy array

.loc lets us index the data in an array-like way, but for explicit index and column names

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

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

Bringing DataFrames closer to 2D arrays: loc; iloc; ix

.iloc lets us index the underlying DataFrame array as if it was a NumPy array

.loc lets us index the data in an array-like way, but for explicit index and column names

.ix is a hybrid of the two (*Deprecated* in recent versions)

```
data.ix[:3, :'pop']
```

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

Deprecated in recent pandas versions

Bringing DataFrames closer to 2D arrays: loc; iloc; ix

.iloc lets us index the underlying DataFrame array as if it was a NumPy array

.loc lets us index the data in an array-like way, but for explicit index and column names

.ix is a hybrid of the two

Any of the NumPy data access tricks can be used with these, e.g.

```
data.loc[data.density > 100, ['pop', 'density']]
```

	pop	density
New York	19651127	139.076746
Florida	19552860	114.806121

Bringing DataFrames closer to 2D arrays: loc; iloc; ix

.iloc lets us index the underlying DataFrame array as if it was a NumPy array

.loc lets us index the data in an array-like way, but for explicit index and column names

.ix is a hybrid of the two

Any of the NumPy data access tricks can be used with these, e.g.

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

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

Bringing DataFrames closer to 2D arrays: loc; iloc; ix

Summary

```
df.iloc[0]   # First row of a data frame
df.iloc[i]   #(i+1)th row
df.iloc[-1] # Last row
```

```
df.iloc[:, 0]   # First column
df.iloc[:, -1] # Last column
```

```
df.iloc[0:7]        #First 7 rows
df.iloc[:, 0:2]     #First 2 columns
df.iloc[1:3, 0:2]   #Second through third rows and first 2 columns
df.iloc[[0,5], [1,3]]  #1st and 6th rows and 2nd and 4th columns
```

Useful indexing tips for DataFrames

Although ***indexing*** refers to columns, ***slicing*** refers to rows:

```
data['Florida':'Illinois']
```

	area	pop	density
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

With slicing we can also refer to rows by number rather than index:

```
data[3:]
```

	area	pop	density
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

Direct masking is interpreted row-wise, rather than column-wise:

```
data[data.density>100]
```

	area	pop	density
New York	141297	19651127	139.076746
Florida	170312	19552860	114.806121

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

Today

1. Multimodal data & Pandas
2. Basic data structures: Series and DataFrames
3. Selecting data in Series and DataFrames
4. **Ufuncs in data frames**

Ufuncs with Pandas

Reminder: do you remember any Ufuncs from NumPy?

NumPy: performs quick element-wise operations

Pandas: inherits much of this functionality from NumPy

With some twists!

Ufuncs with Pandas

Unary operations (e.g. negation and trigonometric functions): Operations in Pandas will preserve index and column labels in the output

Binary operations (e.g. addition and multiplication): Pandas will automatically align indices when passing objects

Unary functions in NumPy

In NumPy:

```
z = -x
```

	x		z
	20		-20
	80		-80
-	90	→	-90
	10		-10

Unary functions and Index Preservation in Pandas

As Pandas is designed to work with NumPy, any NumPy ufunc will work on Series and DataFrames:

Creating a **Series**:

```
import pandas as pd
import numpy as np
```

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

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

Creating a **DataFrame**:

```
df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
df
```

	A	B	C	D
0	6	9	2	6
1	7	4	3	7
2	7	2	5	4

Unary functions and Index Preservation in Pandas

If we apply a NumPy ufunc on either of these objects:
Another Pandas object with **preserved indices**:

Calculation on a **Series**:

```
np.exp(ser)
```

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

Calculation on a **DataFrame**:

```
np.sin(df * np.pi / 4)
```

	A	B	C	D
0	-1.000000	7.071068e-01	1.000000	-1.000000e+00
1	-0.707107	1.224647e-16	0.707107	-7.071068e-01
2	-0.707107	1.000000e+00	-0.707107	1.224647e-16

Binary ufunc operations in NumPy

In NumPy:

z = x / y

x		y		z
20		2		10
80	/	20	→	4
90		10		9
10		5		2

Ufuncs in Pandas

In Pandas Series:

population_density = population/area

```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                  'California': 423967}, name='area')
population = pd.Series({'California': 38332521, 'Texas': 26448193,
                        'New York': 19651127}, name='population')
```

What do you observe with respect to the indices of the two DataFrames?

Ufuncs in Pandas

In Pandas Series:

population_density = population/area

area

Alaska	1723337
Texas	695662
California	423967

/

population

California	38332521
Texas	26448193
New York	19651127

Ufuncs in Pandas

In Pandas Series:

population_density = population/area

area

Alaska	1723337
Texas	695662
California	423967

/

population

California	38332521
Texas	26448193
New York	19651127

→

population_density

Alaska	NaN
Texas	38.018740
California	90.413926
New York	NaN

Ufuncs in Pandas

In Pandas Series:

population_density = population/area

area	
Alaska	1723337
Texas	695662
California	423967

/

population	
California	38332521
Texas	26448193
New York	19651127

→

population_density	
Alaska	NaN
Texas	38.018740
California	90.413926
New York	NaN

Union of indices of the two input arrays

```
area.index | population.index
```

```
Index(['Alaska', 'California', 'New York', 'Texas'], dtype='object')
```

Ufuncs in Pandas

When we divide the two DataFrames to compute population density:

```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                  'California': 423967}, name='area')
population = pd.Series({'California': 38332521, 'Texas': 26448193,
                        'New York': 19651127}, name='population')
```

Result: *union* of indices of the two inputs:

```
population / area
```

```
Alaska              NaN
California    90.413926
New York            NaN
Texas         38.018740
dtype: float64
```

How do we align indexes in DataFrames?

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

	A	B
0	1	11
1	5	1

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

	B	A	C
0	4	0	9
1	5	8	0
2	9	2	6

```
A + B
```

What will the output be??

How do we align indexes in DataFrames?

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

	A	B
0	1	11
1	5	1

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

	B	A	C
0	4	0	9
1	5	8	0
2	9	2	6

```
A + B
```

	A	B	C
0	1.0	15.0	NaN
1	13.0	6.0	NaN
2	NaN	NaN	NaN

- Indices are aligned correctly irrespective of their original order!
- Missing columns or indices in either dataframe are assigned NaN values in the output dataframe

Recap: Indexing

So far: one dimensional and two dimensional data (*Series / DataFrames*)

Often, we need to store high-dimensional data: indexed by more than two keys

Next weeks:

Hierarchical Indexing: (multi-indexing): incorporates multiple index levels within one single index

Dealing with missing values

Aligning datasets

Today Summary

1. Multimodal data & Pandas
2. Basic data structures: Series and DataFrames
3. Selecting data in Series and DataFrames
4. Ufuncs in data frames