UNIVERSITÄT BERN

# 6. Analysing specialized datasets with Python

Athina Tzovara

University of Bern

Athina.Tzovara@unibe.ch

# Today

1. Exploring datasets

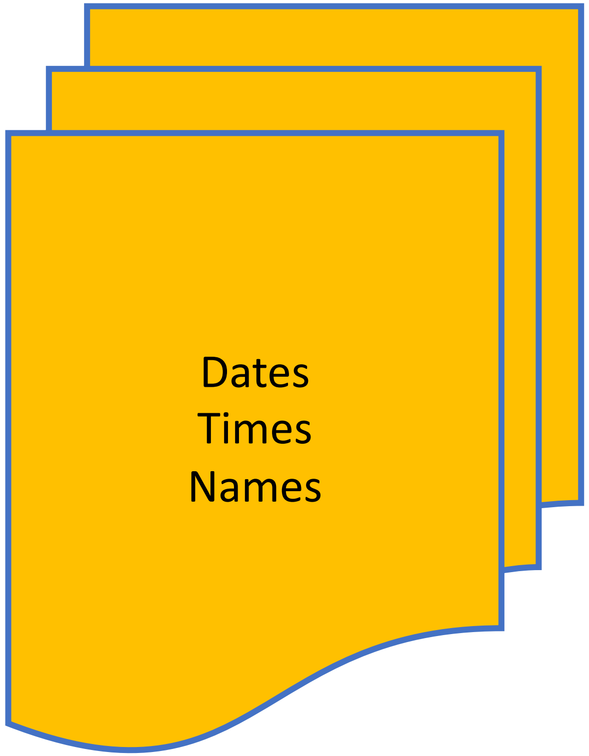
2. Vectorized string operations

3. Working with time series data

4. Examples

# Main question

How do we generate reports for our data in an elegant way?



# How do we summarize a dataset with python?

In 1D NumPy arrays: we can perform aggregations
Similar, for Pandas Series:

```
import numpy as np
import pandas as pd

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

```
0    0.374540
1    0.950714
2    0.731994
3    0.598658
4    0.156019
dtype: float64
```

```
ser.sum()
```

```
2.811925491708157
```

```
ser.mean()
```

```
0.5623850983416314
```

# How do we summarize a dataset with python?

For DataFrames: aggregates will return results within each column (default)
Or within each row (`axis = 'columns'`)

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

| | A | B |
|---|---|---|
| 0 | 0.155995 | 0.020584 |
| 1 | 0.058084 | 0.969910 |
| 2 | 0.866176 | 0.832443 |
| 3 | 0.601115 | 0.212339 |
| 4 | 0.708073 | 0.181825 |

```
df.mean()
```

```
A    0.477888
B    0.443420
dtype: float64
```

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

```
0    0.088290
1    0.513997
2    0.849309
3    0.406727
4    0.444949
dtype: float64
```

# Describing a dataset with python

`DataFrame.describe()` will return a summary of aggregate measures for a DataFrame:

```
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape

planets.dropna().describe()
```

| | number | orbital_period | mass | distance | year |
|---|---|---|---|---|---|
| **count** | 498.00000 | 498.000000 | 498.000000 | 498.000000 | 498.000000 |
| **mean** | 1.73494 | 835.778671 | 2.509320 | 52.068213 | 2007.377510 |
| **std** | 1.17572 | 1469.128259 | 3.636274 | 46.596041 | 4.167284 |
| **min** | 1.00000 | 1.328300 | 0.003600 | 1.350000 | 1989.000000 |
| **25%** | 1.00000 | 38.272250 | 0.212500 | 24.497500 | 2005.000000 |
| **50%** | 1.00000 | 357.000000 | 1.245000 | 39.940000 | 2009.000000 |
| **75%** | 2.00000 | 999.600000 | 2.867500 | 59.332500 | 2011.000000 |
| **max** | 6.00000 | 17337.500000 | 25.000000 | 354.000000 | 2014.000000 |

# Describing a dataset with python

`DataFrame.describe()` will return a summary of aggregate measures for a DataFrame:

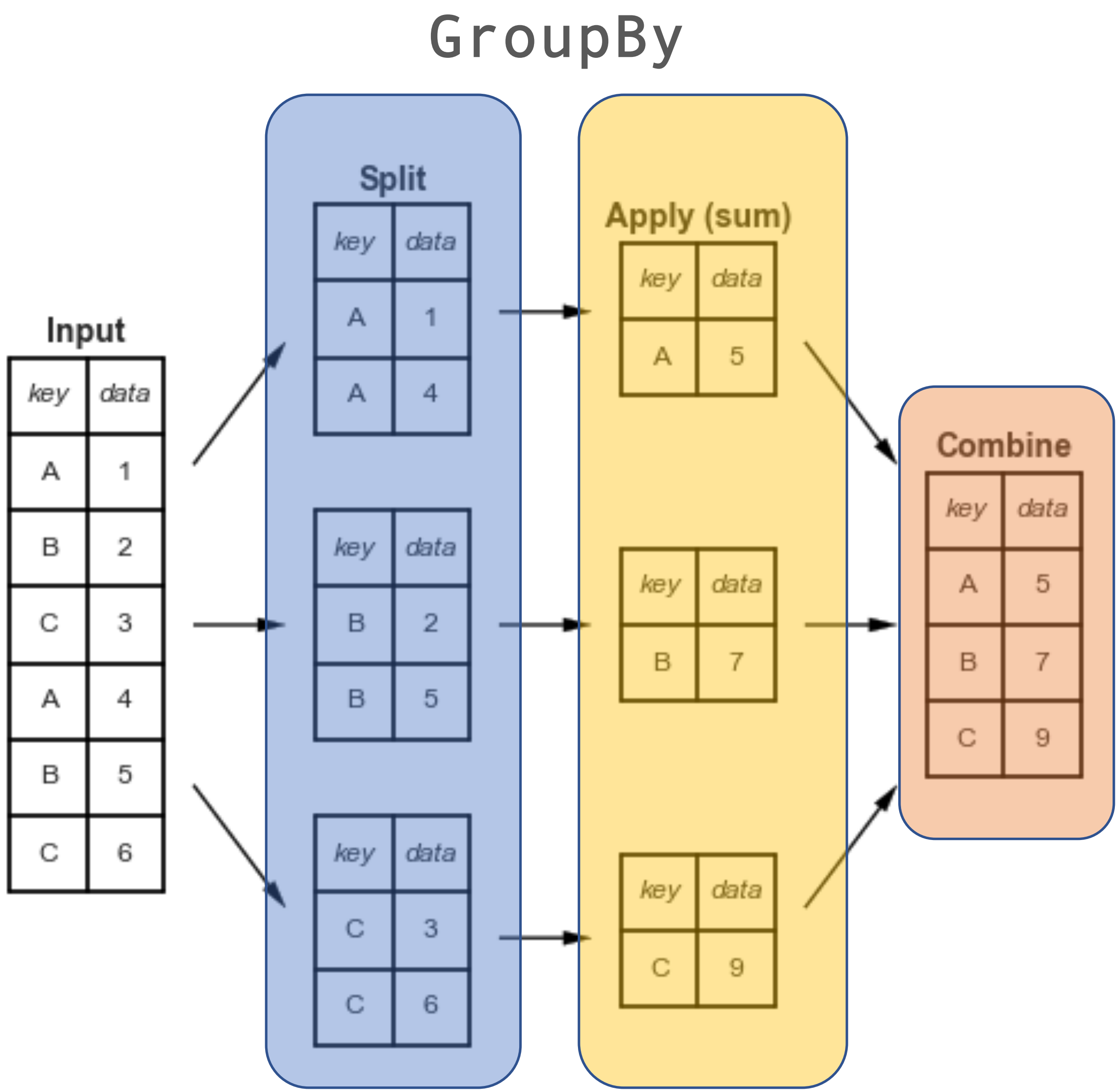
```
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape

planets.dropna().describe()
```

| | number | orbital_period | mass | distance | year |
|---|---|---|---|---|---|
| count | 498.00000 | 498.000000 | 498.000000 | 498.000000 | 498.000000 |
| mean | 1.73494 | 835.778671 | 2.509320 | 52.068213 | 2007.377510 |
| std | 1.17572 | 1469.128259 | 3.636274 | 46.596041 | 4.167284 |
| min | 1.00000 | 1.328300 | 0.003600 | 1.350000 | 1989.000000 |
| 25% | 1.00000 | 38.272250 | 0.212500 | 24.497500 | 2005.000000 |
| 50% | 1.00000 | 357.000000 | 1.245000 | 39.940000 | 2009.000000 |
| 75% | 2.00000 | 999.600000 | 2.867500 | 59.332500 | 2011.000000 |
| max | 6.00000 | 17337.500000 | 25.000000 | 354.000000 | 2014.000000 |

| 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 |

# GroupBy: Split, Apply, Combine




1. **Split**: breaking up a DataFrame and re-grouping
2. **Apply**: computing some function on individual groups
3. **Combine**: merging results of operations

# GroupBy: Split, Apply, Combine

Example of `GroupBy()`

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])
df
```

| | key | data |
|---|---|---|
| 0 | A | 0 |
| 1 | B | 1 |
| 2 | C | 2 |
| 3 | A | 3 |
| 4 | B | 4 |
| 5 | C | 5 |

DataFrame where we will apply groupping

Grouping can be done by passing the name of the desired key column

# GroupBy: Split, Apply, Combine

Example of `GroupBy()`

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])
df
```

| | key | data |
|---|---|---|
| **0** | A | 0 |
| **1** | B | 1 |
| **2** | C | 2 |
| **3** | A | 3 |
| **4** | B | 4 |
| **5** | C | 5 |

```
df.groupby('key')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fadb8042780>
```

DataFrame where we will apply groupping

Grouping can be done by passing the name of the desired key column

`DataFrameGroupBy` object is returned
*No operation until the aggregation is specified*

# GroupBy: Split, Apply, Combine

## Example of GroupBy()

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])
df
```

| | key | data |
|---|---|---|
| **0** | A | 0 |
| **1** | B | 1 |
| **2** | C | 2 |
| **3** | A | 3 |
| **4** | B | 4 |
| **5** | C | 5 |

DataFrame where we will apply groupping

Grouping can be done by passing the name of the desired key column

```
df.groupby('key')
```

<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fadb8042780>

`DataFrameGroupBy` object is returned
*No operation until the aggregation is specified*

```
df.groupby('key').sum()
```

| key | data |
|---|---|
| **A** | 3 |
| **B** | 5 |
| **C** | 7 |

Applying an aggregate to the `DataFrameGroupBy` object

# GroupBy: Example

**Data**: https://www.zoology.ubc.ca/biol548/workshop-graphics.html

```
mammals = pd.read_csv('mammals.csv')
```

```
df.head()
```

| | continent | status | order | family | genus | species | mass.grams |
|---|---|---|---|---|---|---|---|
| **0** | AF | extant | Artiodactyla | Bovidae | Addax | nasomaculatus | 70000.3 |
| **1** | AF | extant | Artiodactyla | Bovidae | Aepyceros | melampus | 52500.1 |
| **2** | AF | extant | Artiodactyla | Bovidae | Alcelaphus | buselaphus | 171001.5 |
| **3** | AF | extant | Artiodactyla | Bovidae | Ammodorcas | clarkei | 28049.8 |
| **4** | AF | extant | Artiodactyla | Bovidae | Ammotragus | lervia | 48000.0 |

1. Import **DataFrame** with data on body mass of various mammals

```
df.groupby('status').count()
```

| | continent | order | family | genus | species | mass.grams |
|---|---|---|---|---|---|---|
| **status** | | | | | | |
| **extant** | 4688 | 5388 | 5388 | 5388 | 5388 | 4061 |
| **extinct** | 164 | 242 | 232 | 242 | 242 | 237 |
| **historical** | 83 | 84 | 84 | 84 | 84 | 44 |
| **introduction** | 17 | 17 | 17 | 17 | 17 | 17 |

2. GroupBy extinction status, and count number of mammals per order/family/genus, etc…

# GroupBy: Example

**Data**: https://www.zoology.ubc.ca/biol548/workshop-graphics.html

```
df.groupby('status')['mass.grams'].mean()
```

```
status
extant          175821.796035
extinct         657086.319831
historical      143548.938636
introduction    179746.852941
Name: mass.grams, dtype: float64
```

3. Compute `mean` mass according to extinction status

```
for (method, group) in df.groupby('status'):
    print("{0:30s} shape={1}".format(method, group.shape))
```

```
extant                          shape=(5388, 7)
extinct                         shape=(242, 7)
historical                      shape=(84, 7)
introduction                    shape=(17, 7)
```

4. Compute `mean` mass according to extinction status

`groupby` supports iteration over groups

# Grouping and aggregating

`GroupBy` also has `aggregate()` `filter()` `transform()` and `apply()` methods

`.aggregate()` performs similar operations as GroupBy but provides additional flexibility

Takes as input: string; function; or list; and computes all aggregates at once:

```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data1': range(6),
                   'data2': rng.randint(0, 10, 6)},
                   columns = ['key', 'data1', 'data2'])
df
```

| | key | data1 | data2 |
|---|---|---|---|
| **0** | A | 0 | 5 |
| **1** | B | 1 | 0 |
| **2** | C | 2 | 3 |
| **3** | A | 3 | 3 |
| **4** | B | 4 | 7 |
| **5** | C | 5 | 9 |

```
df.groupby('key').aggregate(['min', np.median, max])
```

| | data1 | | | data2 | | |
|---|---|---|---|---|---|---|
| | **min** | **median** | **max** | **min** | **median** | **max** |
| **key** | | | | | | |
| **A** | 0 | 1.5 | 3 | 3 | 4.0 | 5 |
| **B** | 1 | 2.5 | 4 | 0 | 3.5 | 7 |
| **C** | 2 | 3.5 | 5 | 3 | 6.0 | 9 |

# Grouping and aggregating

We can also use `.aggregate()` with a dictionary that maps column names to operations:

```
df.groupby('key').aggregate({'data1': 'min',
                             'data2': 'max'})
```

| | data1 | data2 |
|---|---|---|
| **key** | | |
| **A** | 0 | 5 |
| **B** | 1 | 7 |
| **C** | 2 | 9 |

# Filtering of data

`.filter()` lets us filter, or drop, data based on the group properties
We can specify a function for this filter.
*e.g. selecting groups with standard deviation larger than some critical value:*

```
def filter_func(x):
    return x['data2'].std() > 4

display('df', "df.groupby('key').std()",
        "df.groupby('key').filter(filter_func)")
```

df

| | key | data1 | data2 |
|---|---|---|---|
| 0 | A | 0 | 5 |
| 1 | B | 1 | 0 |
| 2 | C | 2 | 3 |
| 3 | A | 3 | 3 |
| 4 | B | 4 | 7 |
| 5 | C | 5 | 9 |

df.groupby('key').std()

| | data1 | data2 |
|---|---|---|
| key | | |
| A | 2.12132 | 1.414214 |
| B | 2.12132 | 4.949747 |
| C | 2.12132 | 4.242641 |

df.groupby('key').filter(filter_func)

| | key | data1 | data2 |
|---|---|---|---|
| 1 | B | 1 | 0 |
| 2 | C | 2 | 3 |
| 4 | B | 4 | 7 |
| 5 | C | 5 | 9 |

# Transformations

With `.transform()` we can return a transformed version of our full dataset, to recombine

The output and input will have the same shape

Example: we can re-center the data to zero-mean:

```
df.groupby('key').transform(lambda x: x - x.mean())
```

| | data1 | data2 |
|---|---|---|
| **0** | -1.5 | 1.0 |
| **1** | -1.5 | -3.5 |
| **2** | -1.5 | -3.0 |
| **3** | 1.5 | -1.0 |
| **4** | 1.5 | 3.5 |
| **5** | 1.5 | 3.0 |

# Apply

The `.apply()` method can apply a function to group results

Input can be a DataFrame, and output a Pandas object, e.g. DataFrame; Series

Example: we can normalize the first column by the sum of the second:

```python
def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x

display('df', "df.groupby('key').apply(norm_by_data2)")
```

df

| | key | data1 | data2 |
|---|---|---|---|
| **0** | A | 0 | 5 |
| **1** | B | 1 | 0 |
| **2** | C | 2 | 3 |
| **3** | A | 3 | 3 |
| **4** | B | 4 | 7 |
| **5** | C | 5 | 9 |

df.groupby('key').apply(norm_by_data2)

| | key | data1 | data2 |
|---|---|---|---|
| **0** | A | 0.000000 | 5 |
| **1** | B | 0.142857 | 0 |
| **2** | C | 0.166667 | 3 |
| **3** | A | 0.375000 | 3 |
| **4** | B | 0.571429 | 7 |
| **5** | C | 0.416667 | 9 |

# Apply

The `.apply()` method can apply a function to group results
Input can be a DataFrame, and output a Pandas object, e.g. DataFrame; Series
Example: we can normalize the first column by the sum of the second:

```
def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x

display('df', "df.groupby('key').apply(norm_by_data2)")
```

df

| | key | data1 | data2 |
|---|---|---|---|
| **0** | A | 0 | 5 |
| **1** | B | 1 | 0 |
| **2** | C | 2 | 3 |
| **3** | A | 3 | 3 |
| **4** | B | 4 | 7 |
| **5** | C | 5 | 9 |

df.groupby('key').apply(norm_by_data2)

| | key | data1 | data2 | |
|---|---|---|---|---|
| **0** | A | 0.000000 | 5 | 0/8 |
| **1** | B | 0.142857 | 0 | 1/7 |
| **2** | C | 0.166667 | 3 | |
| **3** | A | 0.375000 | 3 | |
| **4** | B | 0.571429 | 7 | |
| **5** | C | 0.416667 | 9 | |

```
df.groupby('key').sum()
```

| key | data1 | data2 |
|---|---|---|
| **A** | 3 | 8 |
| **B** | 5 | 7 |
| **C** | 7 | 12 |

```
df
```

| | key | data1 | data2 |
|---|---|---|---|
| **0** | A | 0 | 5 |
| **1** | B | 1 | 0 |
| **2** | C | 2 | 3 |
| **3** | A | 3 | 3 |
| **4** | B | 4 | 7 |
| **5** | C | 5 | 9 |

# Example of grouping

We can easily use `groupby()` `unstack()` `fillna()` to count the number of planets by method and decade:

```
decade = 10 * (planets['year'] // 10)
decade = decade.astype(str) + 's'
decade.name = 'decade'
planets.groupby(['method', decade])['number'].sum().unstack().fillna(0)
```

| decade | 1980s | 1990s | 2000s | 2010s |
|---|---|---|---|---|
| **method** | | | | |
| **Astrometry** | 0.0 | 0.0 | 0.0 | 2.0 |
| **Eclipse Timing Variations** | 0.0 | 0.0 | 5.0 | 10.0 |
| **Imaging** | 0.0 | 0.0 | 29.0 | 21.0 |
| **Microlensing** | 0.0 | 0.0 | 12.0 | 15.0 |
| **Orbital Brightness Modulation** | 0.0 | 0.0 | 0.0 | 5.0 |
| **Pulsar Timing** | 0.0 | 9.0 | 1.0 | 1.0 |
| **Pulsation Timing Variations** | 0.0 | 0.0 | 1.0 | 0.0 |
| **Radial Velocity** | 1.0 | 52.0 | 475.0 | 424.0 |
| **Transit** | 0.0 | 0.0 | 64.0 | 712.0 |
| **Transit Timing Variations** | 0.0 | 0.0 | 0.0 | 9.0 |

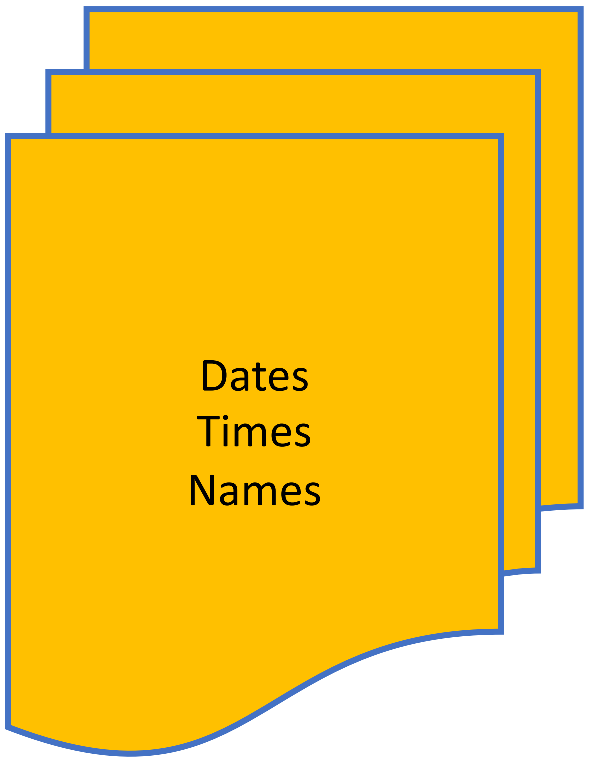
# Additional way to summarize datasets: Pivot tables

**How do we abstract a dataset and generate a ‘report’-like output?**

Pivot Tables vs. Groupby()

Pivot Tables: multidimensional version of Groupby()

Split ; apply; combine : over a two-dimensional grid



# Example of a summary table

One possibility is via `GroupBy()`
Example:

We examine body mass per extinction status:

```
df.groupby('status')['mass.grams'].mean()
```

```
status
extant          175821.796035
extinct         657086.319831
historical      143548.938636
introduction    179746.852941
Name: mass.grams, dtype: float64
```

*Quite straightforward, provides some insight*

# Example of a summary table

One possibility is via `GroupBy()`

What if we want to group by different variables, e.g. extinction status & continent, we select 'mass.grams' and unstack to reveal multi-dimensionality:

```
df.groupby(['status', 'continent'])['mass.grams'].aggregate('mean').unstack()
```

| continent | AF | AUS | Af | EA | Insular | Oceanic | SA |
|---|---|---|---|---|---|---|---|
| status | | | | | | | |
| extant | 31794.525730 | 17484.882609 | NaN | 36413.194572 | 12431.589629 | 8.568480e+06 | 4126.675293 |
| extinct | 970038.461538 | 188355.555556 | NaN | NaN | 95177.912000 | NaN | 985889.131579 |
| historical | 147513.750000 | 2819.045455 | NaN | 302625.000000 | 327198.407692 | NaN | NaN |
| introduction | NaN | 179746.852941 | NaN | NaN | NaN | NaN | NaN |

*Quite complicated code !*

# The alternative: Pivot tables

More readable approach; same result:

Variable we want to compute

```
df.pivot_table('mass.grams', index = 'status', columns = 'continent')
```

| continent | AF | AUS | EA | Insular | Oceanic | SA |
|---|---|---|---|---|---|---|
| **status** | | | | | | |
| **extant** | 31794.525730 | 17484.882609 | 36413.194572 | 12431.589629 | 8.568480e+06 | 4126.675293 |
| **extinct** | 970038.461538 | 188355.555556 | NaN | 95177.912000 | NaN | 985889.131579 |
| **historical** | 147513.750000 | 2819.045455 | 302625.000000 | 327198.407692 | NaN | NaN |
| **introduction** | NaN | 179746.852941 | NaN | NaN | NaN | NaN |

Columns

Index

# Multi-level pivot tables

We can have pivot tables at multiple levels.
For example: we want to compute the mean body mass per extinction status, separately for low and high body mass species.

**1. We start by binning our dataframe:**

```
mass = pd.cut(df['mass.grams'], [10000, 20000, 100000])
print(mass)
```

```
0       (20000.0, 100000.0]
1       (20000.0, 100000.0]
2                       NaN
3       (20000.0, 100000.0]
4       (20000.0, 100000.0]
                ...
5726                    NaN
5727                    NaN
5728    (20000.0, 100000.0]
5729                    NaN
5730                    NaN
Name: mass.grams, Length: 5731, dtype: category
Categories (2, interval[int64]): [(10000, 20000] < (20000, 100000]]
```

Range of cutoff values

# Multi-level pivot tables

We can have pivot tables at multiple levels.
For example: we want to compute the mean body mass per extinction status, separately for low and high body mass species.

**2. We apply the bin in our pivot table:**

```
df.pivot_table('mass.grams', ['continent', mass])
```

bin information

| | | mass.grams |
|---|---|---|
| **continent** | **mass.grams** | |
| **AF** | **(10000, 20000]** | 14440.868571 |
| | **(20000, 100000]** | 50949.577358 |
| **AUS** | **(10000, 20000]** | 13872.727273 |
| | **(20000, 100000]** | 55163.782051 |
| **EA** | **(10000, 20000]** | 14305.961290 |
| | **(20000, 100000]** | 51796.737736 |
| **Insular** | **(10000, 20000]** | 14403.512500 |
| | **(20000, 100000]** | 61935.124000 |
| **Oceanic** | **(20000, 100000]** | 58237.500000 |
| **SA** | **(10000, 20000]** | 14744.422222 |
| | **(20000, 100000]** | 47817.094595 |

# Additional options

We have multiple options for pivot tables:

```
df.pivot_table(values=None, index=None, columns=None,
               aggfunc='mean', fill_value=None,
               margins=False, dropna=True, margins_name='All',
               observed=False, sort=True)
```

*Dealing with missing data*
*Selecting aggregate functions*
*Sorting…*

# Additional options

For example, by default missing values will be dropped. If we unselect that:

```
df.pivot_table('mass.grams', ['continent', mass], dropna = False)
```

| | | mass.grams |
|---|---|---|
| **continent** | **mass.grams** | |
| **AF** | **(10000, 20000]** | 14440.868571 |
| | **(20000, 100000]** | 50949.577358 |
| **AUS** | **(10000, 20000]** | 13872.727273 |
| | **(20000, 100000]** | 55163.782051 |
| **Af** | **(10000, 20000]** | NaN |
| | **(20000, 100000]** | NaN |
| **EA** | **(10000, 20000]** | 14305.961290 |
| | **(20000, 100000]** | 51796.737736 |
| **Insular** | **(10000, 20000]** | 14403.512500 |
| | **(20000, 100000]** | 61935.124000 |
| **Oceanic** | **(10000, 20000]** | NaN |
| | **(20000, 100000]** | 58237.500000 |
| **SA** | **(10000, 20000]** | 14744.422222 |
| | **(20000, 100000]** | 47817.094595 |

# Additional options

For example, we can compute total values with `margins`
or, change the aggregate function:

```
df.pivot_table(['mass.grams', 'continent'], ['status', mass],
               aggfunc='count', margins = True)
```

| | | continent | mass.grams |
|---|---|---|---|
| **status** | **mass.grams** | | |
| **extant** | **(10000, 20000]** | 98.0 | 101.0 |
| | **(20000, 100000]** | 165.0 | 186.0 |
| **extinct** | **(10000, 20000]** | 8.0 | 10.0 |
| | **(20000, 100000]** | 46.0 | 58.0 |
| **historical** | **(10000, 20000]** | 4.0 | 4.0 |
| | **(20000, 100000]** | 6.0 | 6.0 |
| **introduction** | **(20000, 100000]** | 6.0 | 6.0 |
| **All** | | 3644.0 | 3644.0 |

# Example

Data from Centers for Disease Control (CDC):
https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv

```
births = pd.read_csv('births.csv')
```

```
births.head()
```

| | year | month | day | gender | births |
|---|---|---|---|---|---|
| 0 | 1969 | 1 | 1.0 | F | 4046 |
| 1 | 1969 | 1 | 1.0 | M | 4440 |
| 2 | 1969 | 1 | 2.0 | F | 4454 |
| 3 | 1969 | 1 | 2.0 | M | 4548 |
| 4 | 1969 | 1 | 3.0 | F | 4548 |

# Example: we summarize the data

We can create a pivot table to explore the data.

For example, we can get the sum of births by decade and gender:

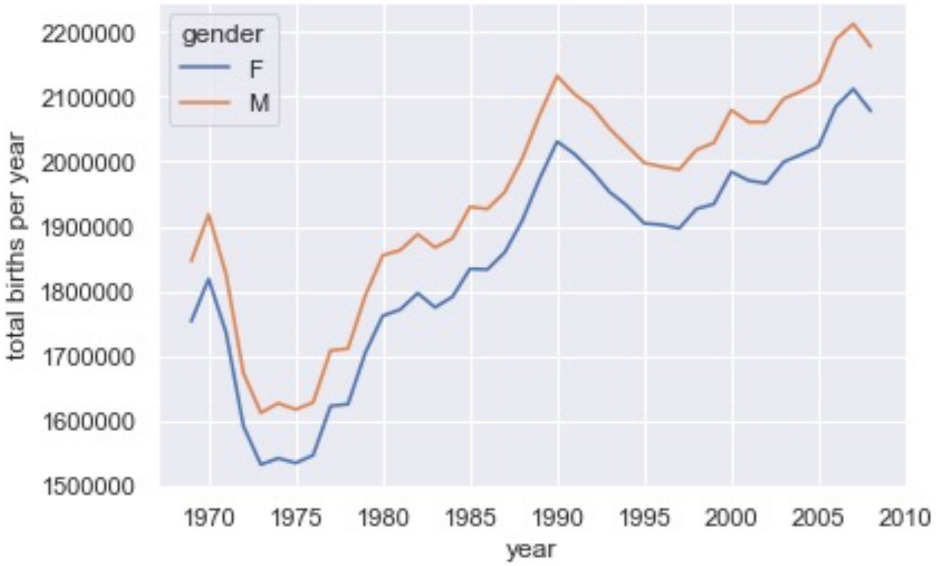
```
births['decade'] = 10 * (births['year'] // 10)
births.pivot_table('births', index='decade',
                   columns='gender', aggfunc='sum')
```

| gender | F | M |
|---|---|---|
| **decade** | | |
| **1960** | 1753634 | 1846572 |
| **1970** | 16263075 | 17121550 |
| **1980** | 18310351 | 19243452 |
| **1990** | 19479454 | 20420553 |
| **2000** | 18229309 | 19106428 |

# Example: we summarize the data

We can also plot the number of births per year, using built-in plotting tools in pandas:

```
%matplotlib inline
import matplotlib.pyplot as plt
sns.set()  # use Seaborn styles
births.pivot_table('births', index='year',
    columns='gender', aggfunc='sum').plot()
plt.ylabel('total births per year');
```



# Today

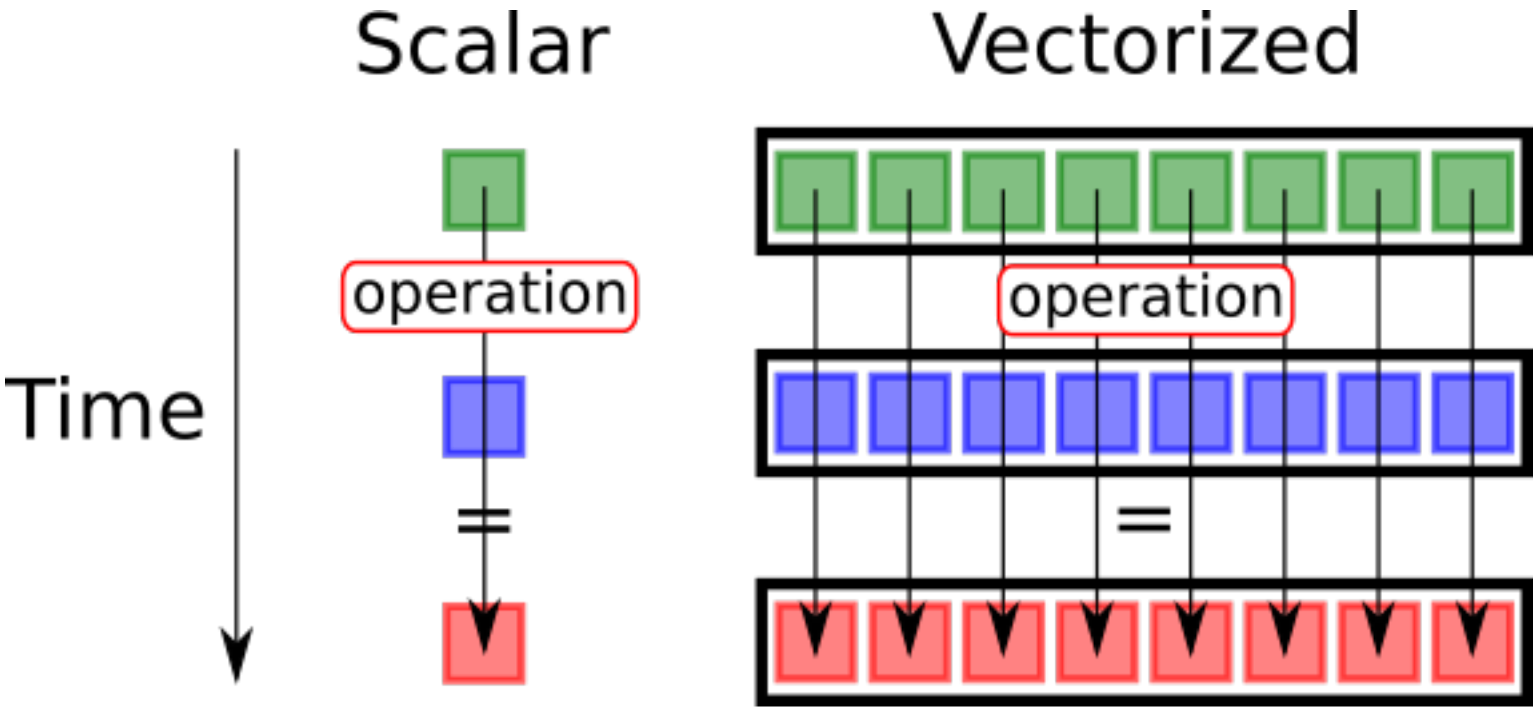
1. Exploring datasets

2. **Vectorized string operations**

3. Working with time series data

4. Examples

# Reminder: Vectorization of operations in Python




Vectorization **simplifies** the syntax of operations

No need to specify size / shape of an array; just the **operation**

# Vectorization for numerical data in python

NumPy and Pandas can generalize arithmetic operations in a quick and efficient way.

For example:

```
import numpy as np
x = np.array([2, 3, 5, 7, 11, 13])
x * 2
```

```
array([ 4,  6, 10, 14, 22, 26])
```

# Vectorization for string arrays in python?

For arrays of strings, NumPy does not provide a simplified access

We need to be more verbose and creative:

```
data = ['peter', 'Paul', 'MARY', 'gUIDO']
[s.capitalize() for s in data]
```

```
['Peter', 'Paul', 'Mary', 'Guido']
```

# Vectorization for string arrays in python?

For arrays of strings, NumPy does not provide a simplified access

However, there are several drawbacks:
e.g. it is not straightforward how to deal with missing data:

```
data = ['peter', 'Paul', 'MARY', 'gUIDO', None]
[s.capitalize() for s in data]
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-108-f2fbc0e54d67> in <module>()
      1 data = ['peter', 'Paul', 'MARY', 'gUIDO', None]
----> 2 [s.capitalize() for s in data]

<ipython-input-108-f2fbc0e54d67> in <listcomp>(.0)
      1 data = ['peter', 'Paul', 'MARY', 'gUIDO', None]
----> 2 [s.capitalize() for s in data]

AttributeError: 'NoneType' object has no attribute 'capitalize'
```

# Vectorization for string arrays in python?

**Alternative**:

Handle arrays of strings via Pandas

- ✓ Address vectorized string operations
- ✓ Handle missing data via `str` attribute

```
import pandas as pd
names = pd.Series(data)
names
```

```
0    peter
1     Paul
2     MARY
3    gUIDO
4     None
dtype: object
```

```
names.str.capitalize()
```

```
0    Peter
1     Paul
2     Mary
3    Guido
4     None
dtype: object
```

# Methods for vectorized string operations

Methods for Pandas `str` similar to Python string methods:

| | | | |
|---|---|---|---|
| `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()` |

# Methods for vectorized string operations

Methods for Pandas `str` similar to Python string methods

Some methods return a `Series` of strings:

```
monte = pd.Series(['Graham Chapman', 'John Cleese',
                   'Terry Gilliam',
                   'Eric Idle', 'Terry Jones',
                   'Michael Palin'])
```

```
monte.str.lower()
```

```
0    graham chapman
1       john cleese
2     terry gilliam
3         eric idle
4       terry jones
5     michael palin
dtype: object
```

# Methods for vectorized string operations

Methods for Pandas `str` similar to Python string methods

Other methods return a `Series` of numerical data:

```
monte.str.len()
```

```
0    14
1    11
2    13
3     9
4    11
5    13
dtype: int64
```

# Methods for vectorized string operations

Methods for Pandas `str` similar to Python string methods

Other methods return a `Series` of Boolean values:

```
monte.str.startswith('T')
```

```
0    False
1    False
2     True
3    False
4     True
5    False
dtype: bool
```

# Methods for vectorized string operations

Methods for Pandas `str` similar to Python string methods

Other methods return a `Series` of compound values:

```
monte.str.split()
```

```
0    [Graham, Chapman]
1       [John, Cleese]
2     [Terry, Gilliam]
3         [Eric, Idle]
4       [Terry, Jones]
5     [Michael, Palin]
dtype: object
```

# Methods using regular expressions

Several methods accept regular expressions
They examine the content of each string element

Same conventions of 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 |

# Methods using regular expressions

Example: we can extract the first name from each entry of our Series,

→Asking for a contiguous group of characters at the beginning of each element:

```
monte.str.extract('([A-Za-z]+)', expand=False)
```

+ one or more occurrences of preceding element

```
0     Graham
1       John
2      Terry
3       Eric
4      Terry
5    Michael
dtype: object
```

# Methods using regular expressions

→Find all names that start and end with a consonant
making use of regular expressions for start of string ^ and end of string $

```
monte.str.findall(r'^[^AEIOU].*[^aeiou]$')
```

```
0    [Graham Chapman]
1                  []
2     [Terry Gilliam]
3                  []
4       [Terry Jones]
5     [Michael Palin]
dtype: object
```

* zero or more occurrences of preceding element
. Matches any character except a newline
^ Matches start of the string
$ Matches the end of the string
[^ ] ndicates a set of characters except the characters listed inside []

https://docs.python.org/3/library/re.html

# Other general methods

We also have other methods that enable interesting operations:

| 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 |

# Vectorized item access and slicing

With `get()` and `slice()` operations we can have vectorized element access from each array

Example: get a slice of the first three characters of each array:

```
monte.str[0:3]
```

```
0    Gra
1    Joh
2    Ter
3    Eri
4    Ter
5    Mic
dtype: object
```

# Vectorized item access and slicing

Example: get the first character of each item in our Series.

Indexing can also be achieved via:

```
monte.str.get(0)
```

```
0    G
1    J
2    T
3    E
4    T
5    M
dtype: object
```

**OR**

```
monte.str[0]
```

```
0    G
1    J
2    T
3    E
4    T
5    M
dtype: object
```

# Vectorized item access and slicing

With `get()` and `slice()` operations we can also access elements of arrays returned by `split()`

Example: extract the last name of **each entry** by combining `split()` and `get()`:

```
monte.str.split().str.get(-1)
```

```
0     Chapman
1      Cleese
2     Gilliam
3        Idle
4       Jones
5       Palin
dtype: object
```

# Vectorized item access and indicator variables

We can use `get_dummies()` when our data have a column with a coded indicator

Example:information in the form of codes,
A="born in America"
B="born in the United Kingdom"
C="likes cheese"
D="likes spam"

```
full_monte = pd.DataFrame({'name': monte,
                           'info': ['B|C|D',
                                    'B|D', 'A|C',
                                    'B|D', 'B|C',
                                    'B|C|D']})
full_monte
```

| | name | info |
|---|---|---|
| 0 | Graham Chapman | B\|C\|D |
| 1 | John Cleese | B\|D |
| 2 | Terry Gilliam | A\|C |
| 3 | Eric Idle | B\|D |
| 4 | Terry Jones | B\|C |
| 5 | Michael Palin | B\|C\|D |

# Vectorized item access and indicator variables

We can use `get_dummies()` when our data have a column with a coded indicator

`get_dummies()` lets us quickly split out the indicators variables, e.g.:

```
full_monte['info'].str.get_dummies('|')
```

| | A | B | C | D |
|---|---|---|---|---|
| **0** | 0 | 1 | 1 | 1 |
| **1** | 0 | 1 | 0 | 1 |
| **2** | 1 | 0 | 1 | 0 |
| **3** | 0 | 1 | 0 | 1 |
| **4** | 0 | 1 | 1 | 0 |
| **5** | 0 | 1 | 1 | 1 |

# Vectorized item access and indicator variables

More information on text data:

**https://pandas.pydata.org/pandas-docs/stable/user_guide/text.html**

# Today

1. Exploring datasets

2. Vectorized string operations

3. **Working with time series data**

4. Examples

# Working with time data

**UTC**

***Coordinated Universal Time***

*Longitude of 0°*


Unix time / epoch

January 1, 1970, at 00:00:00 UTC

Unix time: same rate as UTC

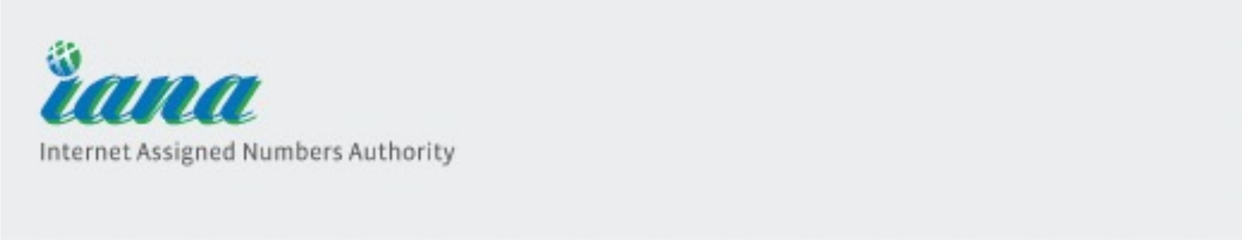
Date and time in UTC: counting number of seconds since the Unix epoch:

```
import time
time.time()
```

```
1635502290.270167
```

# Working with time data

From Unix time: convert information to a human readable format
Time Zone Database: hours and minutes of offset from UTC




## Time Zone Database

The Time Zone Database (often called `tz` or `zoneinfo`) contains code and data that represent the history of local time for many representative locations around the globe. It is updated periodically to reflect changes made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules. Its management procedure is documented in BCP 175: Procedures for Maintaining the Time Zone Database.

Formatting date and time: `YYYY-MM-DD HH:MM:SS`

# Working with time data in Python

`datetime` module: built in in Python

It can perform different functionalities on dates and times

```
from datetime import datetime
datetime(year=2015, month=7, day=4)
```

```
datetime.datetime(2015, 7, 4, 0, 0)
```

# Working with time data in Python

`datetime` module: built-in python

`calendar`: outputs calendars

`time` : time-related functions where dates are not needed

# `datetime` module: built-in python

**We can represent dates and times**

```
from datetime import date, time
```

```
date(year = 2021, month = 10, day = 1)
```

```
datetime.date(2021, 10, 1)
```

```
time(hour = 12, minute = 10, second = 1)
```

```
datetime.time(12, 10, 1)
```

```
datetime.today()
```

```
datetime.datetime(2021, 10, 29, 12, 30, 42, 151493)
```

```
datetime.now()
```

```
datetime.datetime(2021, 10, 29, 12, 30, 45, 941944)
```

**We can obtain current date / time**

# Working with time series data

`dateutil` module: it can parse data from different formats
Contains information on timezones

Example:

```
from dateutil import parser
date = parser.parse("4th of July, 2015")
date
```

```
datetime.datetime(2015, 7, 4, 0, 0)
```

# Working with time series data

`dateutil` module: it can parse data from different formats

With a `datetime` object we have different functionalities, e.g. printing the day of the week:

```
date.strftime('%A')
```

```
'Saturday'
```

**Code for printing days**

# Working with time series data

`dateutil` module: it can parse data from different formats

With a `datetime` object we have different functionalities, e.g. printing the day of the week:

```
date.strftime('%A')
```

```
'Saturday'
```

**Code for printing days**

**Other standard format codes**

```
date.strftime('%B')
```

```
'July'
```

```
date.strftime('%Y')
```

```
'2015'
```

```
date.strftime('%T')
```

```
'00:00:00'
```

*https://docs.python.org/3/library/datetime.html*

# How do we handle multiple time-related entries in python?

`datetime` or `dateutil` are flexible and have intuitive syntax; **however:**
**Working with large arrays of dates and time is not straightforward**

**NumPy**: has native time series data types: `datetime64` data types encode dates as 64-bit integers

Arrays of dates can be represented very compactly

```
import numpy as np
date = np.array('2015-07-04', dtype=np.datetime64)
date
```

```
array('2015-07-04', dtype='datetime64[D]')
```

# `datetime64` arrays for storing time

Advantage: we can perform vectorized operations in `datetime64` data

e.g. via UFuncs:

```
date + np.arange(12)
```

```
array(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07',
       '2015-07-08', '2015-07-09', '2015-07-10', '2015-07-11',
       '2015-07-12', '2015-07-13', '2015-07-14', '2015-07-15'],
      dtype='datetime64[D]')
```

Much faster operations compared to python's `datetime` objects

# `datetime64` vs. `timedelta64`

NumPy does not have a physical quantities system → the `timedelta64` data type complements `datetime64`

```
np.timedelta64(1, 'D')
```

```
numpy.timedelta64(1,'D')
```

1 day

```
np.timedelta64(4, 'h')
```

```
numpy.timedelta64(4,'h')
```

4 hours

```
np.timedelta64('NaT') # Not a Time
```

```
numpy.timedelta64('NaT')
```

Not a Time

```
# How many days have passed from the beginning of the year?
np.datetime64('2021-11-01') - np.datetime64('2021-01-01')
```

```
numpy.timedelta64(304,'D')
```

Datetime calculations

# Trade-off between time resolution and maximum time span

`datetime64` objects are built on a fundamental time unit

Limited to 64 precision --> range of encodable times: $2^{64}$

*Trade-off between time resolution and maximum time span*

*If we want a resolution of nanosecond: we can only encode up to 600 years…*

# Trade-off between time resolution and maximum time span

NumPy will infer the desired time unit from the input. For example:

```
np.datetime64('2015-07-04')
```

```
numpy.datetime64('2015-07-04')
```

Date

```
np.datetime64('2015-07-04 12:00')
```

```
numpy.datetime64('2015-07-04T12:00')
```

Minute-based

```
np.datetime64('2015-07-04 12:59:59.50', 'ns')
```

```
numpy.datetime64('2015-07-04T12:59:59.500000000')
```

Nanosecond-based

# `datetime64` objects

Format codes for `datetime64` `NumPy` objects

| Code | Meaning | Time span (relative) | Time span (absolute) |
|---|---|---|---|
| `Y` | Year | ± 9.2e18 years | [9.2e18 BC, 9.2e18 AD] |
| `M` | Month | ± 7.6e17 years | [7.6e17 BC, 7.6e17 AD] |
| `W` | Week | ± 1.7e17 years | [1.7e17 BC, 1.7e17 AD] |
| `D` | Day | ± 2.5e16 years | [2.5e16 BC, 2.5e16 AD] |
| `h` | Hour | ± 1.0e15 years | [1.0e15 BC, 1.0e15 AD] |
| `m` | Minute | ± 1.7e13 years | [1.7e13 BC, 1.7e13 AD] |
| `s` | Second | ± 2.9e12 years | [ 2.9e9 BC, 2.9e9 AD] |
| `ms` | Millisecond | ± 2.9e9 years | [ 2.9e6 BC, 2.9e6 AD] |
| `us` | Microsecond | ± 2.9e6 years | [290301 BC, 294241 AD] |
| `ns` | Nanosecond | ± 292 years | [ 1678 AD, 2262 AD] |
| `ps` | Picosecond | ± 106 days | [ 1969 AD, 1970 AD] |
| `fs` | Femtosecond | ± 2.6 hours | [ 1969 AD, 1970 AD] |
| `as` | Attosecond | ± 9.2 seconds | [ 1969 AD, 1970 AD] |

*Time zone: automatically set to the local time of the computer executing the code*

# Dealing with date and time in Pandas

Pandas can efficiently deal with time data

Supporting vectorized operations of `NumPy` based on `datetime64`

Working with `Timestamp` objects

`DatetimeIndex` that can index data in Series or DataFrame

```
date = pd.to_datetime("4th of July, 2015")
date
```

```
Timestamp('2015-07-04 00:00:00')
```

## Dealing with date and time in Pandas

Additional operations, similar to `datetime / dateutil`

Example: find day

```
date.strftime('%A')
```

```
'Saturday'
```

Or vectorized operations:

```
date + pd.to_timedelta(np.arange(12), 'D')
```

```
DatetimeIndex(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07',
               '2015-07-08', '2015-07-09', '2015-07-10', '2015-07-11',
               '2015-07-12', '2015-07-13', '2015-07-14', '2015-07-15'],
              dtype='datetime64[ns]', freq=None)
```

# Dealing with date and time in Pandas

Extra advantage: we can index data by timestamps

Example: Series object that has time indexed data

```
index = pd.DatetimeIndex(['2014-07-04', '2014-08-04',
                          '2015-07-04', '2015-08-04'])
data = pd.Series([0, 1, 2, 3], index=index)
data
```

```
2014-07-04    0
2014-08-04    1
2015-07-04    2
2015-08-04    3
dtype: int64
```

# Dealing with date and time in Pandas

## 1. We can use the same indexing patterns as Series

```
data['2014-07-04':'2015-07-04']
```

```
2014-07-04    0
2014-08-04    1
2015-07-04    2
dtype: int64
```

*Obtaining all entries within a range*

## 2. We also have date-only indexing operations

```
data['2015']
```

```
2015-07-04    2
2015-08-04    3
dtype: int64
```

*Obtaining a slice of all data from a year*

# Different Time Series Data Structures

## `Timestamp`

Replacement for Python's `datetime`
Based on numpy's `datetime64` object

## `DatetimeIndex`

Associate Index structure with `Timestamp`

# Different Time Series Data Structures

`pd.to_datetime()` helps parse different date formats:

```
dates = pd.to_datetime([datetime(2015, 7, 3), '4th of July, 2015',
                        '2015-Jul-6', '07-07-2015', '20150708'])
dates
```

```
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
               '2015-07-08'],
              dtype='datetime64[ns]', freq=None)
```

Same data type as numpy

Default converting to
DatetimeIndex

# Different Time Series Data Structures

Easy to convert to a `PeriodIndex`

We need to specify a frequency code

```
dates.to_period('D')
```

```
PeriodIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
             '2015-07-08'],
            dtype='period[D]', freq='D')
```

**Daily frequency**

**period datatype**

*(different from earlier pandas versions)*

**Converting to** `PeriodIndex`

# Different Time Series Data Structures

We can now perform operations like subtraction

Resulting data type: `TimedeltaIndex`

```
dates - dates[0]
```

```
TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'], dtype
='timedelta64[ns]', freq=None)
```

`timedelta64`

# Frequencies and offsets

How do we specify frequency spacing?

| 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 | | |

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

# Different Time Series Data Structures

We can have ranges of time data:

*2 hours (H) 30 minutes (T)*

```
pd.timedelta_range(0, periods=9, freq="2H30T")
```

```
TimedeltaIndex(['0 days 00:00:00', '0 days 02:30:00', '0 days 05:00:00',
                '0 days 07:30:00', '0 days 10:00:00', '0 days 12:30:00',
                '0 days 15:00:00', '0 days 17:30:00', '0 days 20:00:00'],
               dtype='timedelta64[ns]', freq='150T')
```

**9 entries, spaced at 2 hours 30 minutes apart**

# Different Time Series Data Structures

We can have ranges of time data:

***Business Day***

```
from pandas.tseries.offsets import BDay
pd.date_range('2015-07-01', periods=5, freq=BDay())
```

```
DatetimeIndex(['2015-07-01', '2015-07-02', '2015-07-03', '2015-07-06',
               '2015-07-07'],
              dtype='datetime64[ns]', freq='B')
```

**5 entries, separated by business days**

# Dealing with time in Python: Summary

1. Built-in Python: `datetime` / `dateutil`

2. Numpy: `dtype = np.datetime64`

3. Pandas: `pd.to_datetime()`

# Today

1. Exploring datasets
2. Vectorized string operations
3. Working with time series data
4. Examples