

# **Effective Programming Practices for Economists**

## **Data management with pandas**

### **Data types**

Janoś Gabler and Hans-Martin von Gaudecker

# Overview

- Why different data types?
- Converting to efficient dtypes
- Overview of numeric dtypes
- String vs. Categorical
- Working with strings and categoricals

# The need for different data types

Consider the gapminder data

	country	continent	year	life_exp
0	Cuba	Americas	2002	77.16
1	Cuba	Americas	2007	78.27
2	Spain	Europe	2002	79.78
3	Spain	Europe	2007	80.94

```
>>> df.dtypes
```

```
country      string[pyarrow_numpy]  
continent    string[pyarrow_numpy]  
year         int64  
life_exp     float64  
dtype: object
```

- Each column has a dtype
- Enables efficient storage and fast computation
- Dtypes are not always set optimally after loading data

# Benefits of good type representation

- Fast calculations in a low level language
- Access to operations that are only relevant for some types
- Memory efficiency

# Converting to efficient dtypes

```
>>> better_dtypes = {  
...     "country": pd.CategoricalDtype(),  
...     "continent": pd.CategoricalDtype(),  
...     "year": pd.UInt16Dtype(),  
...     "life_exp": pd.Float64Dtype(),  
... }
```

```
>>> df = df.astype(better_dtypes)  
>>> df.dtypes
```

```
country      category  
continent    category  
year         UInt16  
life_exp     Float64  
dtype: object
```

- Depending on how you load your data, the dtypes are not set optimally
- If so, you can create a dictionary that maps columns to the dtypes you want

# Overview of numeric dtypes

Type	Properties
<code>`pd.Int8Dtype()`</code>	Byte (-128 to 127)
<code>`pd.Int16Dtype()`</code>	Integer (-32768 to 32767)
<code>`pd.Int32Dtype()`</code>	Integer (-2147483648 to 2147483647)
<code>`pd.Int64Dtype()`</code>	Integer (-9223372036854775808 to 9223372036854775807)
<code>`pd.UInt8Dtype()`</code>	Unsigned integer (0 to 255)
<code>`pd.UInt16Dtype()`</code>	Unsigned integer (0 to 65535)
<code>`pd.UInt32Dtype()`</code>	Unsigned integer (0 to 4294967295)
<code>`pd.UInt64Dtype()`</code>	Unsigned integer (0 to 18446744073709551615)
<code>`pd.Float64Dtype()`</code>	Double precision float

# String vs. Categorical

- `pd.CategoricalDtype()` is for data that takes values in a fixed and relatively small set of categories
  - Internally stored as small integers
  - Very fast relabeling or resorting of categories
- `pd.StringDtype()` is for actual text data
  - Internally stored as `pyarrow` array
  - Fast string functions similar to methods of Python strings

# Working with strings

```
>>> sr = pd.Series(["Guido", "Tim", "Raymond"])
```

```
>>> sr.str.lower()
```

```
0      guido
1       tim
2    raymond
dtype: string
```

```
>>> sr.str.replace("i", "iii")
```

```
0    Guiiido
1     Tiiim
2    Raymond
dtype: string
```

- The `sr.str` accessor provides access to the string methods
- Vectorized and fast implementations!
- Other examples:
  - `sr.str.len`
  - `sr.str.contains`
  - ...
- See [this tutorial](#) for more string methods



# Working with categoricals

```
>>> cat_type = pd.CategoricalDtype(
...     categories=["low", "middle", "high"],
...     ordered=True,
... )

>>> sr = pd.Series(
...     ["low", "high", "high"],
...     dtype=cat_type,
... )

>>> sr
0    low
1    high
2    high
dtype: category
Categories (3, string): [low < middle < high]
```

- Categories are defined independent of data
  - Protection against invalid categories
  - Good for visualization!
- `sr.cat` accessor provides access to methods
- See [this tutorial](#) for more methods