

Effective Programming Practices for Economists

Data management with pandas

Creating variables

Janoš Gabler and Hans-Martin von Gaudecker

Using numpy math functions

assume that `df` is the gapminder example

```
>>> import numpy as np
>>> df["log_life_exp"] = np.log(df["life_exp"])
>>> df
```

| | country | continent | year | life_exp | log_life_exp |
|---|---------|-----------|------|----------|--------------|
| 0 | Cuba | Americas | 2002 | 77.16 | 4.35 |
| 1 | Cuba | Americas | 2007 | 78.27 | 4.36 |
| 2 | Spain | Europe | 2002 | 79.78 | 4.38 |
| 3 | Spain | Europe | 2007 | 80.94 | 4.39 |

- All functions you'll ever need are implemented:
 - `np.log`
 - `np.exp`
 - `np.sqrt`
 - `np.power`
 - ...
- See [docs](#) for details
- Index is preserved
- Very fast, vectorized implementations

Arithmetic with Series

```
>>> df["gdp_billion"] = df["gdp_per_cap"] * df["pop"] / 1e9
>>> df
```

| | country | year | gdp_per_cap | pop | gdp_billion |
|---|---------|------|-------------|----------|-------------|
| 0 | Cuba | 2002 | 6340.65 | 11226999 | 71.19 |
| 1 | Cuba | 2007 | 8948.10 | 11416987 | 102.16 |
| 2 | Spain | 2002 | 24835.47 | 40152517 | 997.21 |
| 3 | Spain | 2007 | 28821.06 | 40448191 | 1165.76 |

- `*`, `+`, `-`, `/`, ... work as expected
- All calculations are aligned by index
- Not all Series have to come from the same DataFrame or be assigned to a DataFrame

Recoding values

```
>>> df["country_code"] = df["country"].replace(  
...     {"Cuba": "CUB", "Spain": "ESP"}  
... )  
>>> df
```

| | country | continent | year | life_exp | country_code |
|---|---------|-----------|------|----------|--------------|
| 0 | Cuba | Americas | 2002 | 77.16 | CUB |
| 1 | Cuba | Americas | 2007 | 78.27 | CUB |
| 2 | Spain | Europe | 2002 | 79.78 | ESP |
| 3 | Spain | Europe | 2007 | 80.94 | ESP |

- Can be useful to create new variable or fix typos in string variables
- Not super fast, but faster than any looping approach

Vectorized if conditions

```
>>> helper = pd.Series(
...     "rich",
...     index=df.index,
... )

>>> df["income_status"] = helper.where(
    cond=df["gdp_per_cap"] > 10000,
    other="not rich",
)
```

- `pd.Series.where` takes two Series as arguments:
 1. `cond` : Boolean Series determining **where** values are kept
 2. `other` : Series with **values** to be used where `cond` is `False`
- Can express general if conditions using **nested where**
- **Vectorized and fast**

| | country | year | gdp_per_cap | pop | income_status |
|---|---------|------|-------------|----------|---------------|
| 0 | Cuba | 2002 | 6340.65 | 11226999 | not rich |
| 1 | Cuba | 2007 | 8948.10 | 11416987 | not rich |
| 2 | Spain | 2002 | 24835.47 | 40152517 | rich |
| 3 | Spain | 2007 | 28821.06 | 40448191 | rich |

When is it okay to loop?

Over columns:

```
clean = pd.DataFrame()
for var in varlist:
    clean[var] = clean_variable(df[var])
```

- Such a loop is not just ok, it is often the fastest and most readable option
- Accessing and inserting columns is fast
- Even if `clean_variable` is vectorized, it's runtime will completely dominate any loop overhead

Over rows:

- Code example intentionally left blank
- Use the vectorized functions from above instead of loops
- List comprehensions, `df.apply`, `map`, etc. are just python loops in disguise and not faster in this case