

Nástroje pro pokročilou manipulaci s daty

Pomocí knihovny Pandas

Vojtěch MRÁZEK

Fakulta informačních technologií, Vysoké učení technické v Brně
Brno, Czech Republic
mrizek@fit.vutbr.cz



Motivace

- Ukázali jsme si, že datová struktura seznamu (list) Pythonu není příliš efektivní.
- Je možné používat NumPy, co ale s daty ve více sloupcích?
- Proč ale využívat jazyk Pythonu?
 - Je efektivní v oblasti získávání dat,
 - rychlá implementace,
 - data se snadno upravují (čistí),
 - známe jej...
- Co potřebujeme umět dělat s daty:
 - vybírat (*selection*),
 - slučovat a agregovat (*grouping*),
 - třídit (*filtering*),
 - spojovat (*merging, joining, concatenation*),
 - měnit a otáčet tabulky (*reshaping, pivoting, stacking*)

Knihovna Pandas

- Knihovna z roku 2008 (od roku 2011 – 2014 masivní rozšíření)
- Vychází z implementace `data.frame` v jazyce R.
- Výkonná (numpy, numexpr, bottleneck, cython).
- Základní prvek pro datovou analýzu v Pythonu.
- Původně určena pro finanční průmysl.

- Vlastnosti

- Komplexní API
 - **Series** má 417 veřejných metod/atributů
 - **Dataframe** má 425 veřejných metod/atributů
 - Podpora 21 formátů
- Uživatelská základna
 - 5 – 10 milionů uživatelů
 - 2,000 přispěvatelů
 - 24 full-time vývojářů
 - 24,567 commitů na GitHub



Original author(s)	Wes McKinney
Developer(s)	Community
Initial release	11 January 2008; 13 years ago ^[citation needed]
Stable release	1.3.0 ^[1] / 2 July 2021; 3 months ago
Repository	github.com/pandas-dev/pandas
Written in	Python, Cython, C
Operating system	Cross-platform
Type	Technical computing
License	New BSD License
Website	pandas.pydata.org



Základní struktury

- Série indexovaných dat `Series`
- Soubor sérií (tabulka) `DataFrame`

Základní struktury: Series

- **Series je struktura:**
 - jednodimenzionální
 - homogenní
 - oproti NumPy rozšířená o index
 - obsahuje in-memory operace (podobně jako NumPy)
- Díky homogennímu rozložení je možné bez zvýšené náročnosti exportovat data do NumPy pole (a zpět)
- Z implementačního hlediska se u čísel jedná kombinaci
 - NumPy pole
 - asociativního pole (dict)

indexy		hodnoty
A	→	5
B	→	10
C	→	12
D	→	-1
E	→	4

```
s = s1 + s2
```

```
s1["A"]
```

Series - Indexování

- Binární operace se vážou indexy
- Indexy mohou být:
 - unikátní => hashovací pole $O(1)$
 - duplicitní
 - první „B“ se spáruje s prvním „B“
 - druhé „B“ se spáruje s druhým „B“
 - vede k $O(\log N)$ (seřazené) , případně $O(N)$ (musíme projít všechny)
 - seřazené / neseřazené => rychlost vyhledávání
- Jedním z operandů může být i `np.ndarray`, v té chvíli se však není možné operaci vázat indexy.

A	0		B	0		A	NA
B	1		C	1		B	1
C	2	+	D	2	=	C	3
D	3		E	3		D	5
E	4		F	4		E	7
						F	NA

Základní struktury: Series – vytváření

```
# z NumPy nd-array / z pole  
pd.Series(np.random.rand(6))
```

```
#>> 0      0.733545  
#>> 1      0.041656  
#>> 2      0.829660  
#>> 3      0.168460  
#>> 4      0.930627  
#>> 5      0.958148  
#>> dtype: float64
```

```
# Z asociativního pole (dict)  
pd.Series({f"row_{i}": i for i in range(6)})
```

```
#>> row_0      0  
#>> row_1      1  
#>> row_2      2  
#>> row_3      3  
#>> row_4      4  
#>> row_5      5  
#>> dtype: int64
```

```
# Specifikace indexu
```

```
pd.Series(np.random.rand(6), index=['a', 'b', 'c', 'd', 'e', 'f'])
```

```
#>> a      0.802457  
#>> b      0.248572  
#>> c      0.141543  
#>> d      0.100682  
#>> e      0.842970  
#>> f      0.176431  
#>> dtype: float64
```



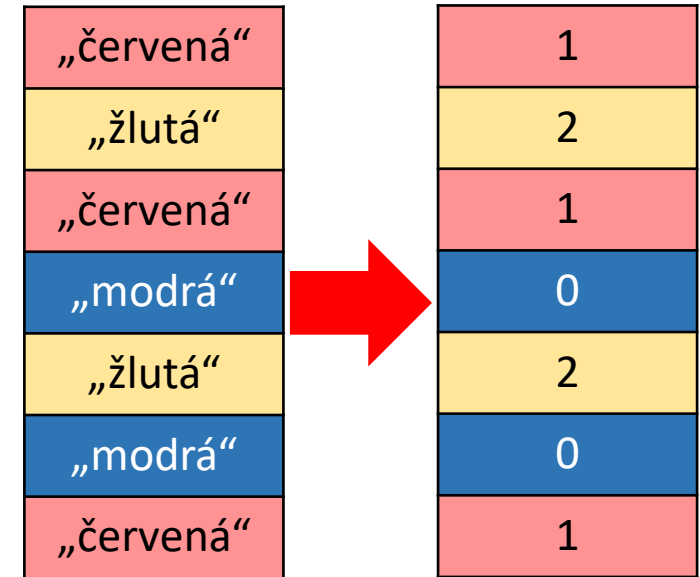
Datové typy

Pandas dtype	Python type	NumPy type	Využití
object	str, mixed	string_, unicode_, mixed types	Textové řetězce nebo mix hodnot
int64	int	int_, int8, int16, int32, int64, uint8, uint16, uint32, uint64	Celá čísla
float64	float	float_, float16, float32, float64	Čísla s plovoucí desetinnou čárkou
bool	bool	bool_	True/False hodnoty
datetime64	NA	datetime64[ns]	Hodnoty data a času (pro časové řady)
timedelta[ns]	NA	NA	Rozdíl mezi dvěma datetime
category	NA	NA	Konečný seznam (textových) hodnot (enum)



Datový typ category

- Analogie enumerate typu v C/C++.
- Vytváří interní mapování číselných hodnot na textové hodnoty `series.astype("category")`
- Pokud se hodnoty opakují, tak se jedná o efektivní ukládání (a zpracování).
- Existuje možnost kategorizovat i numerická data pomocí funkce `pd.cut(x, bins, ...)`.



0	„modrá“
1	„červená“
2	„žlutá“

```
pd.cut([-1, 0, 1, 2, 3, 4, 5], [0, 2, 4])
#>> [NaN, NaN, (0.0, 2.0], (0.0, 2.0], (2.0, 4.0], (2.0, 4.0], NaN]
#>> Categories (2, interval[int64]): [(0, 2] < (2, 4]]
```

```
pd.cut([1, 5, 17, 20, 75], [0, 3, 18, 70, float("inf")],
      labels=["batole", "dítě", "dospělý", "důchodce"])
#>> ['batole', 'dítě', 'dítě', 'dospělý', 'důchodce']
#>> Categories (4, object): ['batole' < 'dítě' < 'dospělý' < 'důchodce']
```

```
s <= "dospělý" # >> array([ True,  True,  True,  True, False])
```

Další poznámky k datovým typům

- Konverze `Series.astype(dtype)` konvertuje datový typ se zachováním hodnot

```
s = pd.Series(["1", 2, 3, 4])
# >> 0      1 (str)
# >> 1      2
# >> 2      3
# >> 3      4
# >> dtype: object
```

```
s.astype("int8")
# >> 0      1 (int)
# >> 1      2
# >> 2      3
# >> 3      4
# >> dtype: int8
```

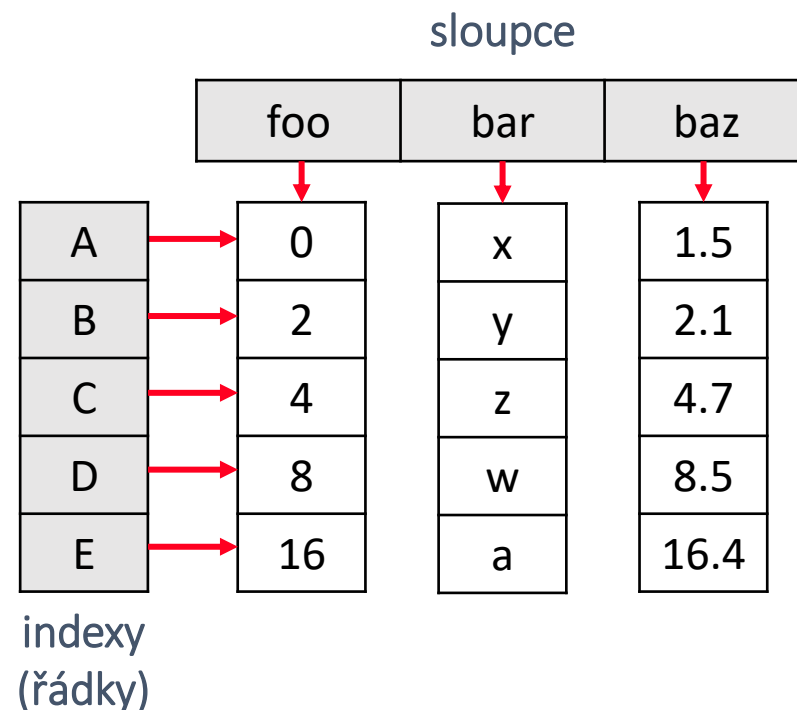
- Analýza paměti `Series.memory_usage(deep=True)`

```
pd.Series(random.choices(["červená", "žlutá", "modrá"],
                        k=1_000_000)).memory_usage(deep=True)
#>> 91 331 578
pd.Series(random.choices(["červená", "žlutá", "modrá"],
                        k=100_000), dtype="category").memory_usage(deep=True)
#>> 100 507
```

- `Int8`, `Int16`, `Int32` a `Int64` (i varianty `UIntX`) umožňují ukládat i NaN.

Základní struktury: DataFrame

- Tabulková reprezentace dat.
- Kombinace více sérií, kde každá série je jeden sloupec.
- Série sdílí jeden index řádků.
- Sloupce jsou také indexované.
- Každý sloupec má stejnou velikost a index, ale může mít různé datové typy.
 - funkce `df.info()` nám řekne detaily o tabulce a její velikost v paměti.
- Celá řada operací
 - jednoduché přidávání sloupců
 - matematické operace mezi sloupci
 - reorganizování dat
 - vyhledávání v datech
 - řazení dat



Uložení dat – v homogenních blocích podle datového typu

DataFrame

	date	number_of_game	day_of_week	v_name	v_league	v_game_number	h_name	h_league	h_game_number	v_score	h_score	length_outs
0	01871054	0	Thu	CL1	na	1	FW1	na	1	0	2	54.0
1	18710505	0	Fri	BS1	na	1	WS3	na	1	20	18	54.0
2	18710506	0	Sat	CL1	na	2	RC1	na	1	12	4	54.0

IntBlock

	0	1	2	3	4	5
0	01871054	0	1	1	0	2
1	18710505	0	1	1	20	18
2	18710506	0	2	1	12	4

ObjectBlock

	0	1	2	3	4
0	Thu	CL1	na	FW1	na
1	Fri	BS1	na	WS3	na
2	Sat	CL1	na	RC1	na

FloatBlock

	0
0	54.0
1	54.0
2	54.0

Limitace knihovny Pandas

- Práce s daty v paměti (in-memory) => použitelné do stovek tisíc až milionu řádků

řádků	Sloupce			
	int64; float64; str(10)	2x int64; 4x float64	4x int64; 4x float64	8x int64; 8x float64
100	7.84 KiB	2.47 KiB	4.81 KiB	9.50 KiB
1_000	77.27 KiB	23.56 KiB	47.00 KiB	93.88 KiB
10_000	771.61 KiB	234.50 KiB	468.88 KiB	937.62 KiB
100_000	7.53 MiB	2.29 MiB	4.58 MiB	9.16 MiB
1_000_000	75.34 MiB	22.89 MiB	45.78 MiB	91.55 MiB
10_000_000	753.40 MiB	228.88 MiB	457.76 MiB	915.53 MiB

DataFrame – vytváření z pole

- Přímočaré vytváření z 2D polí (NumPy i klasický list)
- Typicky specifikujeme názvy sloupců.
- Můžeme určit i indexy, případně se použije výchozí celočíselné indexování

```
pd.DataFrame(np.random.random(size=(6, 4)),
             columns=["col_1", "col_2", "col_3", "col_4"])
#>>      col_1      col_2      col_3      col_4
#>> 0  0.987477  0.544273  0.914710  0.115794
#>> 1  0.987215  0.627147  0.547343  0.871113
#>> 2  0.552167  0.042253  0.281974  0.521386
#>> 3  0.374902  0.370970  0.868235  0.187717
#>> 4  0.668802  0.183973  0.369614  0.554067
#>> 5  0.002002  0.742656  0.262801  0.759456
```

```
pd.DataFrame(np.random.random(size=(6, 4)),
             columns=["col_1", "col_2", "col_3", "col_4"],
             index=["A", "B", "C", "D", "E", "F"])
#>>      col_1      col_2      col_3      col_4
#>> A  0.393709  0.962003  0.172676  0.971163
#>> B  0.491043  0.910839  0.699413  0.145003
#>> C  0.198114  0.596928  0.227296  0.370077
#>> D  0.197594  0.974489  0.370042  0.415160
#>> E  0.364902  0.161861  0.991155  0.399498
#>> F  0.586258  0.925872  0.556381  0.964225
```

DataFrame – vytváření ze sérií

```
s1 = pd.Series(np.random.random(6), name="teplota")
s2 = pd.Series(np.random.random(6), name="vlhkost")
s3 = pd.Series(np.random.random(8), name="oblacnost")
```

```
pd.DataFrame([s1, s2])
```

```
#>>          0          1          2          3          4          5
#>> teplota  0.684456  0.709938  0.023831  0.343873  0.870008  0.026573
#>> vlhkost  0.652168  0.634678  0.744319  0.540563  0.530810  0.375137
```

- Seznam (list) definuje jednotlivé řádky.
- Pomocí asoc. pole (dict) specifikujeme sloupce (možno použít i list nebo ndarray).
- U sérií dojde k zarovnání indexů.

```
pd.DataFrame({"teplota": s1, "vlhkost": s2, "oblacnost": s3})
#>>      teplota  vlhkost  oblacnost
#>> 0  0.684456  0.652168  0.863086
#>> 1  0.709938  0.634678  0.983394
#>> 2  0.023831  0.744319  0.743858
#>> 3  0.343873  0.540563  0.245281
#>> 4  0.870008  0.530810  0.402906
#>> 5  0.026573  0.375137  0.457101
#>> 6         NaN         NaN  0.010125
#>> 7         NaN         NaN  0.244775
```



DataFrame – vytváření z asoc. polí (dict)

```
data = {f"gid_{g}": {
    f"iid_{i}": np.random.randint(0, 10) for i in range(6)
} for g in range(4) }
```

```
#>> {'gid_0': {'iid_0': 2, 'iid_1': 3, 'iid_2': 3, 'iid_3': 9, 'iid_4': 0, 'iid_5': 9},
#>>   'gid_1': {'iid_0': 5, 'iid_1': 1, 'iid_2': 1, 'iid_3': 2, 'iid_4': 9, 'iid_5': 2},
#>>   'gid_2': {'iid_0': 0, 'iid_1': 0, 'iid_2': 9, 'iid_3': 8, 'iid_4': 4, 'iid_5': 6},
#>>   'gid_3': {'iid_0': 4, 'iid_1': 3, 'iid_2': 0, 'iid_3': 9, 'iid_4': 4, 'iid_5': 3}}
```

```
pd.DataFrame.from_dict(data)
```

```
#>>      gid_0  gid_1  gid_2  gid_3
#>> iid_0      2      5      0      4
#>> iid_1      3      1      0      3
#>> iid_2      3      1      9      0
#>> iid_3      9      2      8      9
#>> iid_4      0      9      4      4
#>> iid_5      9      2      6      3
```

- Parametr orient říká, zda hlavní klíče odpovídají sloupcům (defaultní) či řádkovému indexu.

- Podobné chování u `pd.DataFrame.from_tuples`, kde musíme navíc specifikovat názvy vnitřních indexů.

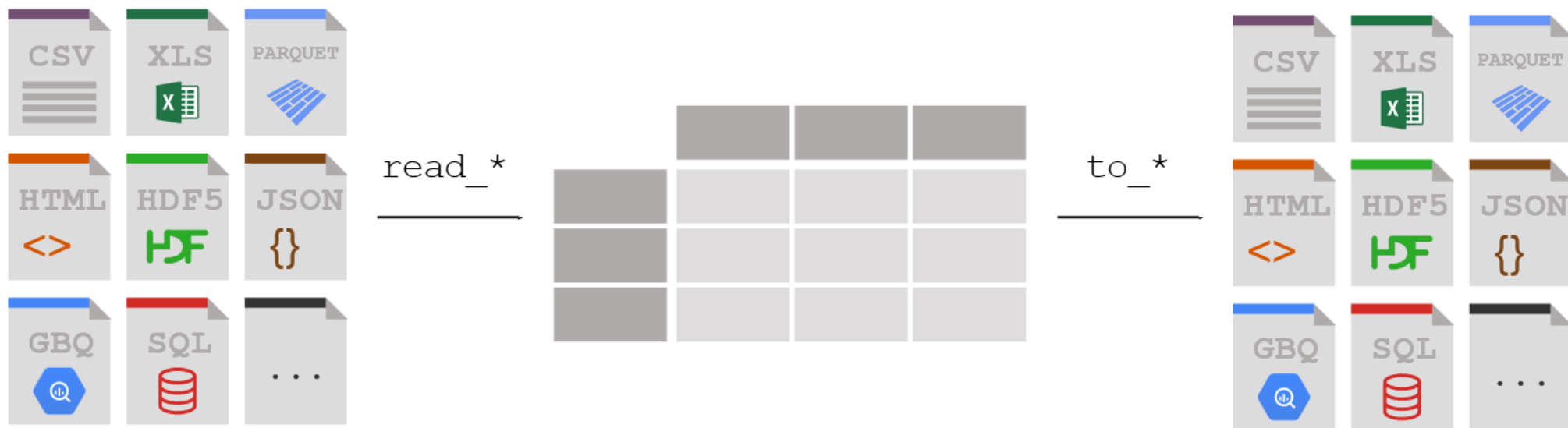
- Často neřešíme indexy řádků, stačí nám potom pole asociativních polí (`list[dict]`)

```
pd.DataFrame.from_dict(data, orient="index")
```

```
#>>      iid_0  iid_1  iid_2  iid_3  iid_4  iid_5
#>> gid_0      2      3      3      9      0      9
#>> gid_1      5      1      1      2      9      2
#>> gid_2      0      0      9      8      4      6
#>> gid_3      4      3      0      9      4      3
```


DataFrame – vytváření ze souborů (a ukládání)

- Nativní podpora 21 formátů a různých kompresí.
- Pro řadu formátů je potřeba instalovat další knihovny (pytables pro HDF5, pyarrow pro Parquet a Feather, xlrd pro XLS, kompresní knihovny, ...).
- Pro optimalizaci přístupu k velkým datům je možné použít tzv. *columnar datové typy* (Parquet, HDF (někdy), Feather, ...) – nemusíme načítat do paměti všechny sloupce, ale jen ty, které potřebujeme.
- Zápis: `pd.read_*` a `df.to_*`, kde `*` je požadovaný datový typ.
- Celá řada formátově-specifických parametrů – viz dokumentace.



Srovnání výkonnosti

■ Pickle

- nativní podpora v Pythonu
- může být problém ve verzích a zpětnou kompatibilitou (lze specifikovat).

■ HDF – formát

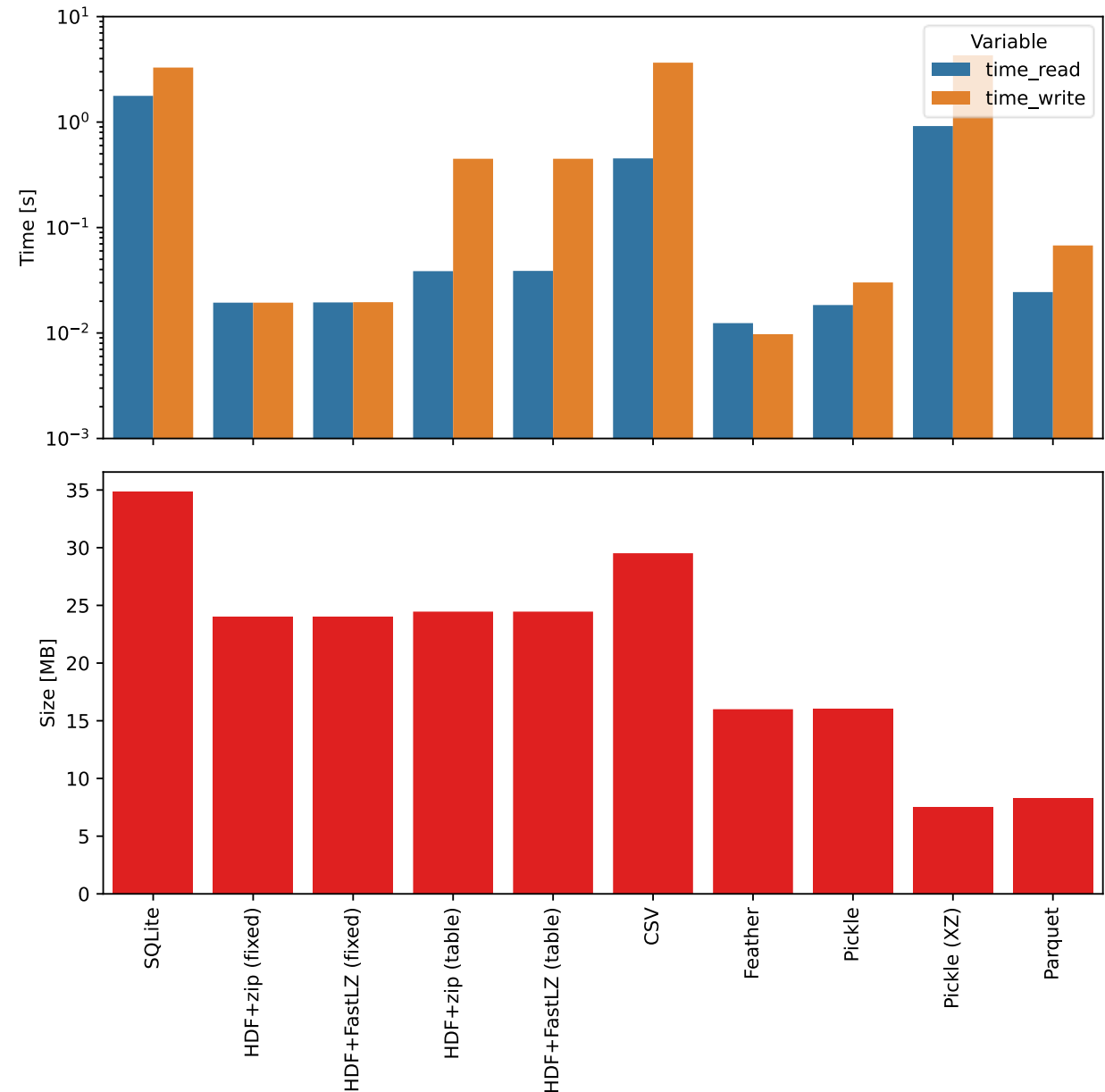
- **fixed**: Fixní formát pro rychlý zápis a čtení, není možné přidávat data (musí se přepsat celý), není *columnar*. Defaultní
- **table**: Tabulkový formát, s horším výkonem, ale je možné přidávat data bez přepsání všeho a je *columnar*.

■ Feather

- formát z nástroje Apache Hadoop
- pro krátkodobé ukládání
- rychlejší, *columnar*.

■ Parquet

- formát z nástroje Apache Arrow
- pro velká data, dlouhodobé ukládání
- menší, pomalejší zápis
- v budoucnu podpora operací in-memory!
- vhodný pro big-data, *columnar*.



Popis experimentů:

https://pandas.pydata.org/docs/user_guide/io.html#io-perf



Adresace dat v DataFrame (df)

Zápis `df[A]`, kde typ A rozhoduje o chování:

- `str -> Series`
 - vrací sloupec
- `List[str] -> DataFrame`
 - vrací skupinu sloupců
- `List[bool] -> DataFrame`
 - vrací vybrané řádky
 - seznam může být `list`, `np.ndarray`, `pd.Series`
- `slice -> DataFrame`
 - indexujeme řádky
 - buď rozsah indexů "row_0" : "row_7" nebo celočíselný rozsah 0 : 6
 - není intuitivní při celočíselných indexech a sloupcích!

	col_0	col_1	col_2	col_3	col_4	col_5	col_6	col_7
row_0	0	1	2	3	4	5	6	7
row_1	8	9	10	11	12	13	14	15
row_2	16	17	18	19	20	21	22	23
row_3	24	25	26	27	28	29	30	31
row_4	32	33	34	35	36	37	38	39
row_5	40	41	42	43	44	45	46	47
row_6	48	49	50	51	52	53	54	55
row_7	56	57	58	59	60	61	62	63

indexy

Poznámka: `df[0]` nemůžeme u této tabulky napsat, ale `df[0:1]` ano – proč?

Jaký může být problém při zápisu `df.col_1`?

Adresace dat v DataFrame (df) - řádková

■ Pomocí atributu `df.loc` (dle názvu)

■ Zápis

`df.loc[řádky, sloupce]`

– řádky

- hodnota (typu indexu řádků)
- seznam (typu indexu řádků)
- bool list
- bool series
- slice (stejného typu, jako indexy)

– sloupce

- hodnota
- seznam
- bool list
- slice (stejného typu, jako hlavičky)

■ Existuje podobný zápis `.at`, který vrací pouze skalární hodnotu a je o kousek rychlejší

■ Pomocí atributu `df.iloc` (podle pořadí)

■ Zápis

`df.iloc[řádky, sloupce]`

– řádky

- celočíselné pořadí
- seznam celých čísel
- bool list
- ~~• bool series~~
- slice (stejného typu, jako indexy)

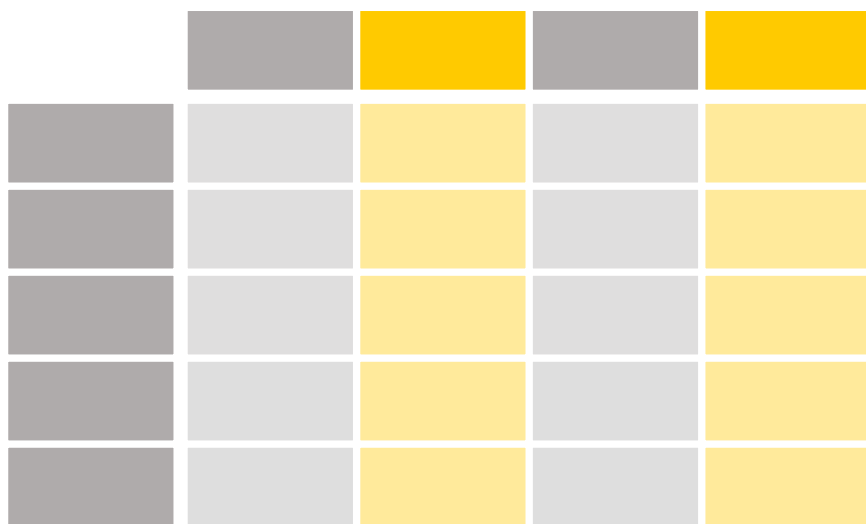
– sloupce

- seznam
- bool list
- slice (celých čísel)

■ Existuje podobný zápis `.iat`, který vrací pouze skalární hodnotu a je o kousek rychlejší

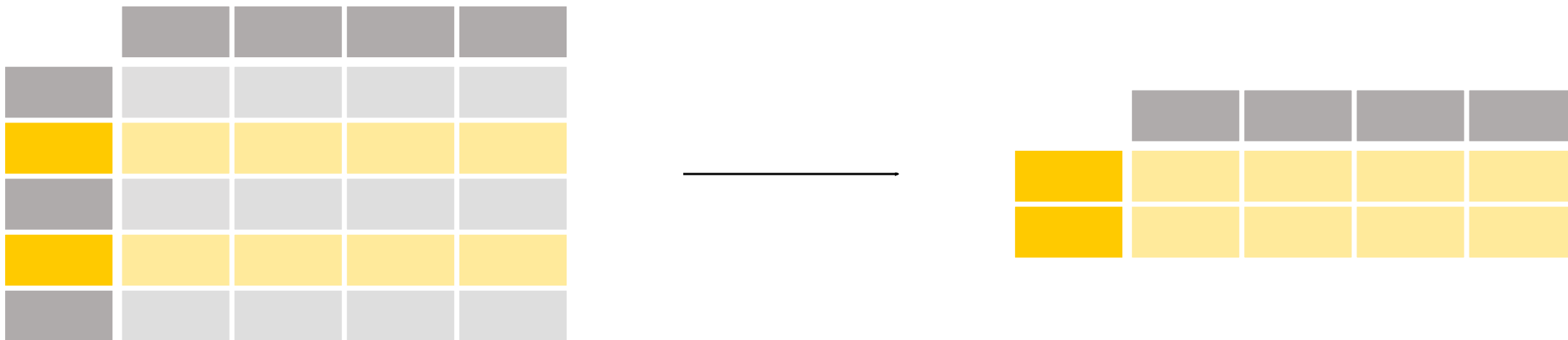


Adresace dat - shrnutí



```
df["nazev_sloupce"] # -> Series  
df[["sloupec1", "sloupec2"]] # -> DataFrame
```

Adresace dat - shrnutí



```
df[df["sloupcec1"] < 42] # -> DataFrame  
df.loc["id_radku"] # -> Series pokud neni duplicitni index  
df.loc["id_radku"] # -> DataFrame pokud je duplicitni index
```

Adresace dat - shrnutí



```
df.loc[["id_radku", "id_radku2"], ["sloupec1", "sloupec2"]]  
df.loc[df["sloupec1"] < 42, ["sloupec2", "sloupec3"]]  
df.loc[df["sloupec1"] < 42, "sloupec1" : "sloupec15"]  
df.iloc[0:5:2 , [1, 2]]
```

Pokročilé indexování

- Použití
 - Agregace dat,
 - Rychlé hledání podle více parametrů,
 - Přeskupování tabulek.
- Pomocí funkce `df.set_index(columns, inplace=True)` můžeme nastavit nový index podle sloupce.
- Sloupec může být jeden, nebo je jich více – tzv. multiindexing.
- Vhodné pro manipulaci s daty, ovšem pro vizualizaci často musíme indexy zrušit pomocí funkce `df.reset_index` (indexy převedeme na sloupce).

```
df = pd.DataFrame({
    "mesto": ["Brno"]*3+["Praha"]*3+["Ostrava"]*3,
    "typ" : ["prumer", "min", "max"] *3,
    "teplota" : [10, 8, 15, 12, 7, 18, 13, 8, 16]})
```

```
#>>      mesto      typ  teplota
#>> 0      Brno  prumer        10
#>> 1      Brno    min         8
#>> 2      Brno    max        15
#>> 3      Praha  prumer        12
```

```
df.set_index(["mesto", "typ"], inplace=True)
```

```
#>>      teplota
#>> mesto  typ
#>> Brno   prumer    10
#>>        min      8
#>>        max    15
#>> Praha  prumer    12
#>>        min      7
#>>        max    18
#>> Ostrava prumer    13
#>>        min      8
#>>        max    16
```


Pokročilé indexování – adresace prvků

- Indexace je pomocí n-tic.

```
df.loc[("Brno", "min")]
#>> teplota      8
#>> Name: (Brno, min), dtype: int64
```

- Nemusíme specifikovat vše

```
df.loc["Ostrava"]
#>>          teplota
#>> typ
#>> prumer      13
#>> min         8
#>> max        16
```

Poznámka: Hodnota „min“ musí být definovaná ve všech položkách prvního indexu!
Pokud není, je vhodnější používat podmínky.

- Indexovat můžeme i rozsahem (pozor na zápis)

```
df.loc[:, "min"]
#>> df.loc[:, "min"]
#>>      ^
#>> SyntaxError: invalid syntax
```

```
df.loc[(slice(None), "min"), :]
#>>          teplota
#>> mesto  typ
#>> Brno   min      8
#>> Praha min      7
```

Pokročilé indexování – transformace dat

- Transformace jednoho řádkového indexu na sloupcový index se provádí pomocí funkce `unstack`

```
df
#>>
#>> mesto typ      teplota
#>> Brno   prumer    10
#>>        min       8
#>>        max      15
#>> Praha prumer    12
#>>        min       7
#>>        max      18
#>> Ostrava prumer   13
#>>        min       8
#>>        max      16
```

```
df.unstack(level="typ")
#>>      teplota
#>> typ      max min prumer
#>> mesto
#>> Brno      15  8    10
#>> Ostrava    16  8    13
#>> Praha     18  7    12
```

Ze sloupcového indexu se nyní stal multiindex!

```
df.unstack(level="typ")["teplota", "min"]
```

```
#>> mesto      Brno Ostrava Praha
#>> typ
#>> max      15     16    18
#>> min       8      8     7
#>> prumer   10     13    12
```

Pokročilé indexování – transformace dat

- Inverzní proces – přesunutí jednoho sloupcového indexu do řádků se provádí pomocí funkce `stack`.

```
df
#>>      teplota
#>> mesto      Brno Ostrava Praha
#>> typ
#>> max          15      16      18
#>> min           8       8       7
#>> prumer        10      13      12
```

```
df.stack(level="mesto")
#>>      teplota
#>> typ      mesto
#>> max      Brno      15
#>>          Ostrava    16
#>>          Praha     18
#>> min      Brno       8
#>>          Ostrava     8
#>>          Praha      7
#>> prumer   Brno      10
#>>          Ostrava    13
#>>          Praha     12
```

```
df.stack(level=0)
#>> mesto      Brno Ostrava Praha
#>> typ
#>> max      teplota      15      16      18
#>> min      teplota       8       8       7
#>> prumer   teplota      10      13      12
```

Nepojmenovaný index => sloupec také nemá jméno

Výběr dat podle podmínek

- Přes boolean lokátor

```
df[df["mesto"] == "Brno"]  
#>> mesto typ teplota  
#>> 0 Brno prumer 10  
#>> 1 Brno min 8  
#>> 2 Brno max 15
```

```
df  
#>> mesto typ teplota  
#>> 0 Brno prumer 10  
#>> 1 Brno min 8  
#>> 2 Brno max 15  
#>> 3 Praha prumer 12  
#>> 4 Praha min 7  
#>> 5 Praha max 18  
#>> 6 Ostrava prumer 13  
#>> 7 Ostrava min 8  
#>> 8 Ostrava max 16
```

- Alternativní zřetězený zápis [query](#)
(složitější parsování, vhodný pro velká data)

```
df.query("mesto == 'Brno'")
```

- Skládání podobně jako v NumPy (&, |, ~ operátory, pozor na závorky!)

```
df[(df["mesto"] == "Brno") & (df["typ"].isin(["min", "max"]))]  
#>> mesto typ teplota  
#>> 1 Brno min 8  
#>> 2 Brno max 15
```

Výběr dat – pokračování

- Další výběr je možný pomocí funkce `where` (nejméně efektivní) – nahrazení ostatních dat pomocí `NaN`

```
df.where(df["mesto"] == "Brno")  
#>> mesto      typ  teplota  
#>> 0  Brno  prumer    10.0  
#>> 1  Brno    min     8.0  
#>> 2  Brno    max    15.0  
#>> 3   NaN   NaN     NaN  
#>> ...  
#>> 8   NaN   NaN     NaN
```

- Kdy vlastně Pandas kopíruje data a kdy vytváří pouze pohled?
 - Nejsme to schopni jednoznačně říct, protože se snaží data co nejvíce využívat mezivýsledky.

```
df["teplota"] # <-- pohled  
df["teplota"].astype("f") # <-- kopie
```

```
df["teplota"][0] = 2 # SettingWithCopyWarning; data se zapíšou do df  
df["teplota"].astype("f")[1] = 2 # zápis bez chyby, výsledky se ztratí
```

Výběr dat – index vs Boolean

- Výběr pomocí indexu je rychlejší než vytvoření Boolean pole (ideálně $O(1)$ vs $O(N)$)
- Jenže vytváření indexu nás také něco stojí!
- Je potřeba volit rozvážně.

```
df = pd.DataFrame({"names": random.choices(
    ["retezec1", "retezec2", "retezec3", "retezec4"], k=1_000_000),
    "values": np.random.normal((100_000))})
```

```
%timeit df[df["names"] == "retezec1"]
```

```
#>> 49.4 ms ± 312 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit dfi = df.set_index("names")
```

```
#>> 13.4 ms ± 53.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit dfi.loc["retezec1"]
```

```
#>> 24.3 ms ± 204 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Přidávání dat

- Přidávání sloupce je přirozená operace, která nevyžaduje přeskupení dat.

```
df["new_column_1"] = "Hodnota" # broadcasting  
df["new_column_2"] = np.random.normal(size=(10)) # musí sedět délka
```

- Přidávání jednoho řádku

- je nutné ignorovat indexy (**append** totiž má připojit i celý **DataFrame** – ekvivalent **pd.concat**).
- vytváří nový **DataFrame**, nejedná se o **inplace** operátor a nemůžeme jej tak ani vyvolat.

```
df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))  
#>>      A  B  
#>> 0    1  2  
#>> 1    3  4
```

```
df = df.append({"A":4, "B": 8}, ignore_index=True)  
#>>      A  B  
#>> 0    1  2  
#>> 1    3  4  
#>> 2    4  8
```

Práce s hodnotami

- binární operace (+, -, *, /, ...)

```
df["c"] = df["a"] + df["b"]  
df["d"] = df["b"]  
df.loc[df["a"] != 0, "d"] = df["b"] / df["a"]
```

- získání unikátních hodnot

```
df["mesto"].unique()  
#>> array(['Brno', 'Praha'], dtype=object)
```

- získání četnosti hodnot

```
df["mesto"].value_counts()  
#>> Brno      3  
#>> Praha     2  
#>> Name: mesto, dtype: int64
```

- statistika `Series.{mean, median, describe, sum, ...}` – více v příští přednášce
- řazení `DataFrame.{sort_values, n_largest, ...}`



Složitější úprava dat pomocí funkce `apply`

- Volání funkce nad všemi daty v sérii či všemi řádky v tabulce (DataFrame).
- Pomalejší, než přímé binární operace, zejména u numerických typů.
- Můžeme volat nad sérií

```
df
#>>      A  B
#>> 0    1  2
#>> 1    3  4
```

```
df["C"] = df["A"].apply(lambda x: x**2) # jednodussi df["A"]**2
#>>      A  B  C
#>> 0    1  2  1
#>> 1    3  4  9
```

- Nebo nad celým rámcem (nutné specifikovat osu)

– `axis = 0` (výchozí) – parametrem funkce je sloupec

```
df.apply(lambda x: x.sum())
#>> A      4; B      6; C     10; D     10; dtype: int64
```

– `axis = 1` – parametrem funkce je každý samostatný řádek

```
df["D"] = df.apply(lambda x: x["A"] + x["B"], axis=1) # jednodussi df["A"] + df["B"]
#>>      A  B  C  D
#>> 0    1  2  1  3
#>> 1    3  4  9  7
```

Složitější úprava dat pomocí funkce `eval`

- Vhodná pro velmi velká data (režie volání parseru příkazu)
- Podpora omezeného množství [operací](#)
- Získání série z existujících

```
df
#>>      A  B  C  D
#>> 0    1  2  1  3
#>> 1    3  4  9  7
```

```
df.eval("A+B")
#>> 0    3
#>> 1    7
#>> dtype: int64
```

- Zápis do dalších sloupců
 - vrací zase nový DataFrame!
 - umožňuje více operací

```
df = df.eval("X = A + B")
#>>      A  B  C  D  X
#>> 0    1  2  1  3  3
#>> 1    3  4  9  7  7
```

```
df = df.eval("""
Y = A+B
Z = A-B
""")
#>>      A  B  C  D  X  Y  Z
#>> 0    1  2  1  3  3  3 -1
#>> 1    3  4  9  7  7  7 -1
```

Accessor objects

- V závislosti na datovém typu uloženého objektu.
- Rychlý přístup k objektům.
- Tento koncept umožňuje snadné rozšíření Pandas o další datové typy (např. GeoPandas (GIS data), cyberpandas (pro IP adresy), ...)
- V některých případech může být `apply` rychlejší.

Data Type	Accessor
Datetime, Timedelta, Period	<code>dt</code>
String	<code>str</code>
Categorical	<code>cat</code>
Sparse	<code>sparse</code>

```
df = pd.DataFrame({"names": random.choices([
    "retezec1", "retezec2", "retezec3", "retezec4"], k=1_000_000),
    "values": np.random.normal(size=(1_000_000))})

%timeit df.names.str.replace("retezec", "string_")
#>> 450 ms ± 6.01 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
%timeit df.names.apply(lambda x: x.replace("retezec", "string_"))
#>> 191 ms ± 2.44 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Změna rozměrů tabulky



```
df = pd.DataFrame({"mesto": ["Brno", "Brno", "Praha", "Praha"],
                  "parametr": ["teplota", "vlhkost", "teplota", "vlhkost"],
                  "hodnota": [15.6, 85, 17.3, 89 ] })

#>> mesto parametr hodnota
#>> 0 Brno teplota 15.6
#>> 1 Brno vlhkost 85.0
#>> 2 Praha teplota 17.3
#>> 3 Praha vlhkost 89.0
pd.pivot(df, columns="parametr", values="hodnota", index="mesto")
#>> parametr teplota vlhkost
#>> mesto
#>> Brno 15.6 85.0
#>> Praha 17.3 89.0
```

Funkce `pivot_table` nám navíc umožňuje zvolit agregační funkci pro různé hodnoty

Změna rozměrů tabulky

```
df = pd.DataFrame({"mesto": ["Brno", "Brno", "Praha", "Praha", "Brno"],  
                  "parametr": ["teplota", "vlhkost", "teplota", "vlhkost", "teplota" ],  
                  "hodnota": [15.6, 85, 17.3, 89, 20 ] })
```

```
#>>      mesto parametr  hodnota  
#>> 0    Brno  teplota    15.6  
#>> 1    Brno  vlhkost    85.0  
#>> 2    Praha teplota    17.3  
#>> 3    Praha vlhkost    89.0  
#>> 4    Brno  teplota    20.0
```

```
pd.pivot(df, columns="parametr", values="hodnota", index="mesto")
```

ValueError: Index contains duplicate entries, cannot reshape

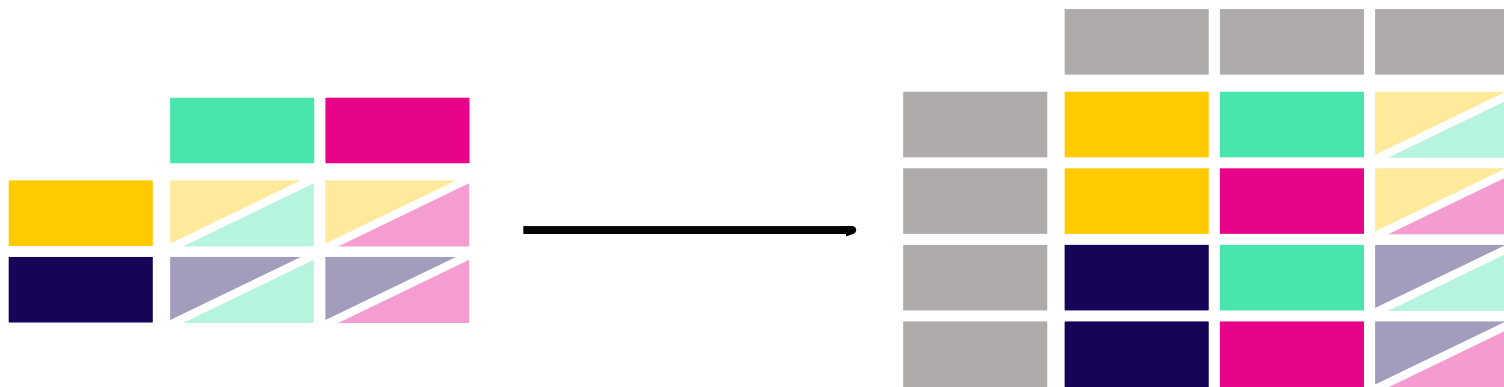
```
pd.pivot_table(df, columns="parametr",  
               values="hodnota", index="mesto",  
               aggfunc="mean")
```

```
#>> parametr  teplota  vlhkost  
#>> mesto  
#>> Brno      17.8      85.0  
#>> Praha     17.3      89.0
```

```
pd.pivot_table(df, columns="parametr",  
               values="hodnota", index="mesto",  
               aggfunc="max")
```

```
#>> parametr  teplota  vlhkost  
#>> mesto  
#>> Brno      20.0      85.0  
#>> Praha     17.3      89.
```

Opačnou funkcí k `pivot` je funkce `melt`



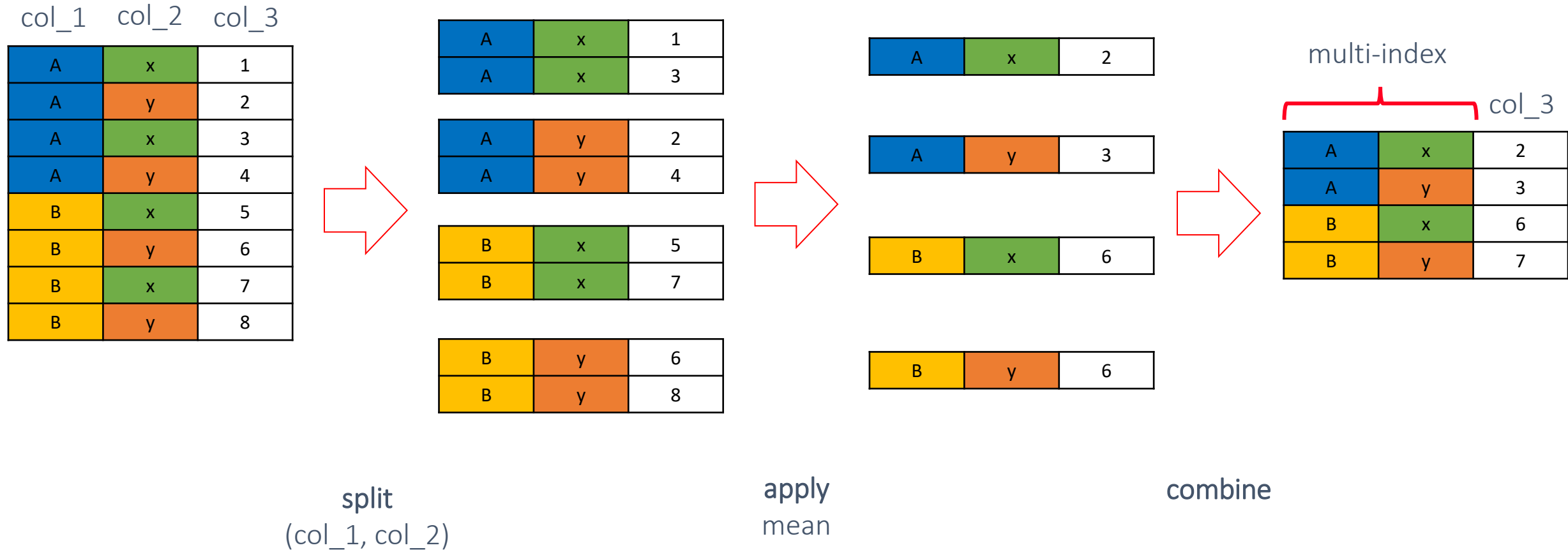
```
df = pd.DataFrame({"mesto": ["Praha", "Brno"], "teplota": [14, 16],
                  "vlhkost": [85.0, 89.0]})
```

```
#>>      mesto  teplota  vlhkost
#>> 0   Praha      14      85.0
#>> 1    Brno      16      89.0
```

```
df.melt(id_vars="mesto", var_name="parametr", value_name="hodnota")
```

```
#>>      mesto parametr  hodnota
#>> 0   Praha  teplota      14.0
#>> 1    Brno  teplota      16.0
#>> 2   Praha  vlhkost      85.0
#>> 3    Brno  vlhkost      89.0
```

Agregace dat



Funkce `groupby`

Funkce `agg`

Seskupení dat pomocí groupby

```
df = pd.DataFrame({"col_1": list("AAAABBBB"), "col_2": list("xy") * 4,  
                  "col_3": np.arange(8), "col_4": np.arange(8)+10})  
  
#>>   col_1 col_2 col_3 col_4  
#>> 0     A     x     0    10  
#>> 1     A     y     1    11  
#>> 2     A     x     2    12  
#>> 3     A     y     3    13  
#>> 4     B     x     4    14  
#>> 5     B     y     5    15  
#>> 6     B     x     6    16  
#>> 7     B     y     7    17
```

Co je návratovou hodnotou funkce groupby?

```
df.groupby(["col_1", "col_2"])  
#>> <pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f00373a71c0>
```


Agregace dat po groupby pomocí agg

- Použijeme agregační funkci na všechny ostatní sloupce

```
df.groupby(["col_1", "col_2"]).agg("mean")  
#>>                col_3  col_4  
#>> col_1 col_2  
#>> A      x      1      11  
#>>      y      2      12  
#>> B      x      5      15  
#>>      y      6      16
```

- Nebo můžeme na každý sloupec použít jinou, není potřeba specifikovat všechny sloupce

```
df.groupby(["col_1", "col_2"]).agg({"col_3": "mean", "col_4": "max"})  
#>>                col_3  col_4  
#>> col_1 col_2  
#>> A      x      1      12  
#>>      y      2      13  
#>> B      x      5      16  
#>>      y      6      17
```

Agregace dat po groupby pomocí agg

- Indexovat můžeme i jedním sloupcem. Kam se ztratil sloupec col_1?

```
df.groupby("col_2").agg("mean")
```

```
#>>          col_3  col_4
```

```
#>> col_2
```

```
#>> x          3      13
```

```
#>> y          4      14
```

```
df.groupby("col_2").agg("last")
```

```
#>>          col_1  col_3  col_4
```

```
#>> col_2
```

```
#>> x          B        6      16
```

```
#>> y          B        7      17
```

- Funkcí můžeme použít více:

```
df.groupby("col_2").agg([np.mean, "last", "max"])
```

```
#>>          col_3          col_4
```

```
#>>          mean last max  mean last max
```

```
#>> col_2
```

```
#>> x          3      6      6      13      16      16
```

```
#>> y          4      7      7      14      17      17
```

} sloupcový multiindex

- Pokud nám (multi-)index neseďí, můžeme se ho zbavit

```
df.groupby("col_2").agg("last").reset_index()
```

```
#>>   col_2 col_1  col_3  col_4
```


```
#>> 0     x     B      6     16
```

```
#>> 1     y     B      7     17
```

Další zpracování dat po groupby

- Není potřeba pokaždé používat funkci `agg`.
- Můžeme přistoupit k jednotlivým sloupcům ve `split` a na ně zavolat libovolnou funkci.
 - Příklad: `cumsum`, `shift`, ...
- Funkce se volá separátně na každý rozdělený rámec po *split*.
- Nativně pak můžeme přiřadit data zpět díky zachování indexů.

```
df
#>>   col_1 col_2 col_3 col_4
#>> 0     A     x     0    10
#>> 1     A     y     1    11
#>> 2     A     x     2    12
#>> 3     A     y     3    13
#>> 4     B     x     4    14
#>> 5     B     y     5    15
#>> 6     B     x     6    16
#>> 7     B     y     7    17
```



```
df["col3_shifted"]=df.groupby("col_1").col_3.shift(-1)
#>>   col_1 col_2 col_3 col_4 col3_shifted
#>> 0     A     x     0    10           1.0
#>> 1     A     y     1    11           2.0
#>> 2     A     x     2    12           3.0
#>> 3     A     y     3    13           NaN
#>> 4     B     x     4    14           5.0
#>> 5     B     y     5    15           6.0
#>> 6     B     x     6    16           7.0
#>> 7     B     y     7    17           NaN
```

Spojování datových rámců

■ Spojení rámců za sebe

- může vést k duplicitním řádkovým indexům, pokud není `ignore_index=True`

```
pd.concat([df1, df2, df3])
```

df1					Result				
	A	B	C	D		A	B	C	D
0	A0	B0	C0	D0	0	A0	B0	C0	D0
1	A1	B1	C1	D1	1	A1	B1	C1	D1
2	A2	B2	C2	D2	2	A2	B2	C2	D2
3	A3	B3	C3	D3	3	A3	B3	C3	D3
df2					4	A4	B4	C4	D4
	A	B	C	D	5	A5	B5	C5	D5
4	A4	B4	C4	D4	6	A6	B6	C6	D6
5	A5	B5	C5	D5	7	A7	B7	C7	D7
6	A6	B6	C6	D6	8	A8	B8	C8	D8
7	A7	B7	C7	D7	9	A9	B9	C9	D9
df3					10	A10	B10	C10	D10
	A	B	C	D	11	A11	B11	C11	D11
8	A8	B8	C8	D8					
9	A9	B9	C9	D9					
10	A10	B10	C10	D10					
11	A11	B11	C11	D11					

Spojování datových rámců

- Spojení rámců za sebe
 - nebo můžeme vytvořit multiindex

```
pd.concat(frames, keys=['x', 'y', 'z'])
```

df1				
	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2				
	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3				
	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result					
		A	B	C	D
x	0	A0	B0	C0	D0
x	1	A1	B1	C1	D1
x	2	A2	B2	C2	D2
x	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
y	5	A5	B5	C5	D5
y	6	A6	B6	C6	D6
y	7	A7	B7	C7	D7
z	8	A8	B8	C8	D8
z	9	A9	B9	C9	D9
z	10	A10	B10	C10	D10
z	11	A11	B11	C11	D11

Spojování datových rámců

- Na základě indexů můžeme spojit rámce i ve směru sloupců

```
pd.concat([df1, df4], axis=1, sort=False)
```

- Definicí typu spojení na `inner` zachováme pouze řádky, které jsou v obou rámcích (2 a 3)

```
pd.concat([df1, df4], axis=1, join='inner')
```

df1					df4				Result							
	A	B	C	D		B	D	F		A	B	C	D	B	D	F
0	A0	B0	C0	D0	2	B2	D2	F2	0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	3	B3	D3	F3	1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	6	B6	D6	F6	2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	7	B7	D7	F7	3	A3	B3	C3	D3	B3	D3	F3
									6	NaN	NaN	NaN	NaN	B6	D6	F6
									7	NaN	NaN	NaN	NaN	B7	D7	F7

Operace obecného spojení datových rámců

```
pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None,  
         left_index=False, right_index=False, sort=True,  
         suffixes=('_x', '_y'), copy=True, indicator=False,  
         validate=None)
```

- Nespojujeme jenom podle indexů, ale podle sloupců (tzv. spojovací klíč).
- Spojovacích klíčů může být více, na obou stranách stejný index.
- Musí být definovaný atribut `on` (v obou rámcích) nebo `*_on` nebo `*_index` pro levý i pravý rámeček.
- V případě duplicitních sloupců se přidají oba sloupce a k názvu se doplní sufix
 - často používáme `suffixes=("", "_y")`
- Můžeme specifikovat atribut `validate` (string, default None)
 - `one_to_one` nebo `1:1` – testuje, zda je spojovací klíč unikátní na obou stranách.
 - `one_to_many` nebo `1:m` – testuje, zda je spojovací klíč unikátní na levé straně.
 - `many_to_one` nebo `m:1` – testuje, zda je spojovací klíč unikátní na pravé straně.
 - `many_to_many` nebo `m:m` – nic netestuje.

Ukázka dat z CHMI

Z historických teplot v Brně-Tuřanech určíme různé ukazatele

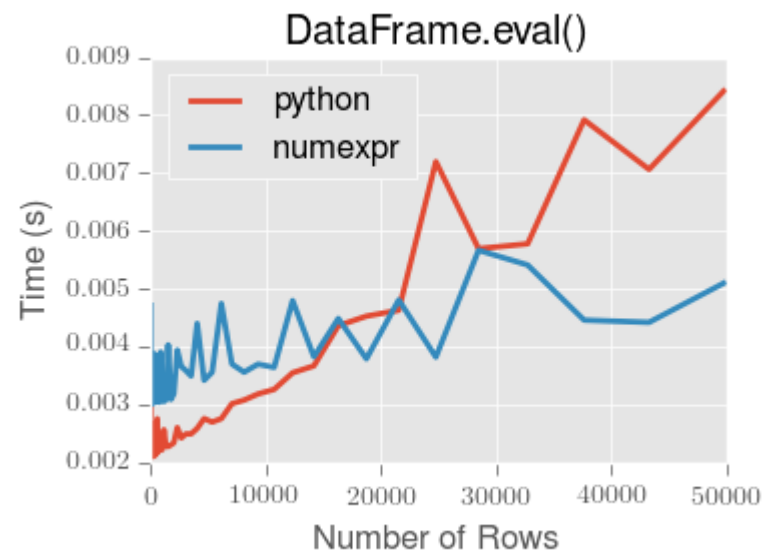
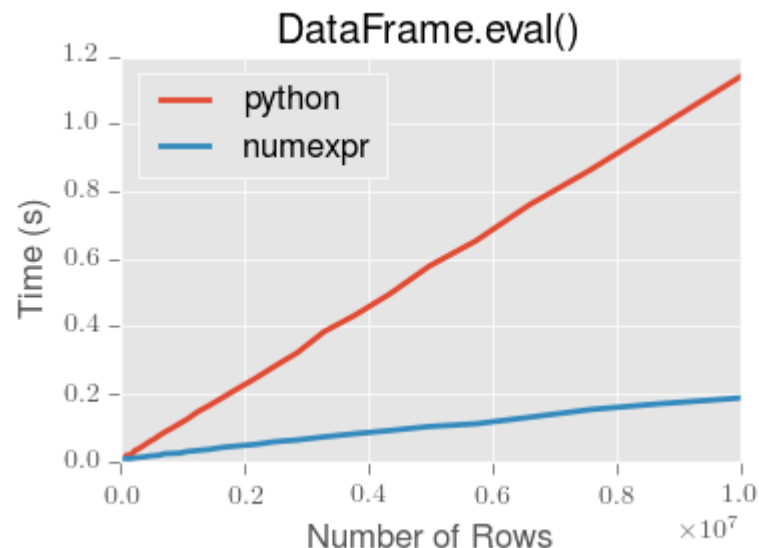


Zadání je k dispozici v podpůrných souborech jako Jupyter notebook. Pro otestování správného pochopení problematiky je doporučeno úkoly reprodukovat.

Optimalizace časových nároků

Má smysl řešit pro velká data!

- Použití dalších nástrojů jako Cython nebo Numba Funkce `eval` vs `apply`



- Zřetězené příkazy

```
(df[['fl_date', 'tail_num', 'dep_time', 'dep_delay']]\n    .dropna()\n    .assign(hour=lambda x: x.dep_time.dt.hour)\n    .query('5 < dep_delay < 600'))
```

Škálování pro velké datasety

- Načítat jen nezbytné množství dat (pouze sloupce, které potřebujeme - viz *columnar* datové formáty)
- Používat efektivní datové typy (category, bitová šířka, omezení PyObject).
- Rozdělení dat do skupin (*chunks*), se kterými pracujeme samostatně.
- Využití dalších paralelních knihoven – např. [Dask](#), který má kompatibilní API.

	Peak memory use	Wallclock time	CPU time
Pandas (chunked)	26 MiB	5.9 seconds	6.1 seconds
Pandas (naive)	334 MiB	6.7 seconds	6.9 seconds
Dask	88 MiB	3.9 seconds	6.5 seconds

But first, it's worth considering *not using pandas*. Pandas isn't the right tool for all situations. If you're working with very large datasets and a tool like PostgreSQL fits your needs, then you should probably be using that. Assuming you want or need the expressiveness and power of pandas, let's carry on

Další zdroje

- Základní příkazy
https://pandas.pydata.org/docs/getting_started/intro_tutorials/index.html
- Přechod z [SQL](#)
- Komplexní uživatelská příručka
https://pandas.pydata.org/docs/user_guide/index.html
- Cheatsheet
https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf