UNIVERSITÄT BERN

5. Working with structured data
Part 2

Athina Tzovara

University of Bern

Athina.Tzovara@unibe.ch

Last week

1. Multimodal data & Pandas

2. Basic data structures: Series and DataFrames

3. Selecting data in Series and DataFrames

4. Ufuncs in data frames

Today

1. Pandas: recap
2. Indexing part 2 (multi-indexing)
3. Working with missing data
4. Concatenating datasets

Reminder: Indexing a series

Example: we want to track data about the population of US states in 2000

```
import pandas as pd
index = ['California',
         'New York',
         'Texas']

populations = [33871648,
               18976457,
               20851820]
pop_2000 = pd.Series(populations, index=index)
pop_2000
```

```
California    33871648
New York      18976457
Texas         20851820
dtype: int64
```

`index`: will contain the indexes with which we can access the population of US states

What if we want to have more than one indexes?

Multi-indexed series: the “bad” way

Example: we want to track data about states from two years (2000 and 2010)

```
index = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]
pop = pd.Series(populations, index=index)
pop
```

```
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
(Texas, 2010)         25145561
dtype: int64
```

It is tempting to use Python tuples as keys where we keep track of state and year

```
my_tuple = (“pen", “pencil", “book")
```

`Tuples:` storing multiple items in one single variable

Multi-indexed series: the “bad” way

Example: we want to track data about states from two years (2000 and 2010)

```
index = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]
pop = pd.Series(populations, index=index)
pop
```

```
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
(Texas, 2010)         25145561
dtype: int64
```

```
pop[('California', 2010):('Texas', 2000)]
```

```
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
dtype: int64
```

Straightforward way: we can use index or slice the series based on the multiple index

Multi-indexed series: the “bad” way

Example: we want to track data about states from two years (2000 and 2010)

```
index = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]
pop = pd.Series(populations, index=index)
pop
```

```
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
(Texas, 2010)         25145561
dtype: int64
```

```
pop[[i for i in pop.index if i[1] == 2010]]
```

```
(California, 2010)    37253956
(New York, 2010)      19378102
(Texas, 2010)         25145561
dtype: int64
```

However, the convenience ends there. For example, if we want to select all values from 2010, we need to be very creative

MultiIndex: **a cleaner way**

Example: we want to track data about states from two years (2000 and 2010)

```
index = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]
pop = pd.Series(populations, index=index)
pop
```

```
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
(Texas, 2010)         25145561
dtype: int64
```

We can create a multi-index directly from the tuple

We can think of multiple *levels* of indexing (state names and years) as levels

```
index = pd.MultiIndex.from_tuples(index)
index
```

```
MultiIndex([('California', 2000),
            ('California', 2010),
            (  'New York', 2000),
            (  'New York', 2010),
            (     'Texas', 2000),
            (     'Texas', 2010)],
           )
```

MultiIndex: **a cleaner way**

We can use the MultiIndex to re-index our series
This creates a hierarchical representation of the data

```
index = pd.MultiIndex.from_tuples(index)
index
```

```
MultiIndex([('California', 2000),
            ('California', 2010),
            (  'New York', 2000),
            (  'New York', 2010),
            (     'Texas', 2000),
            (     'Texas', 2010)],
           )
```

```
pop = pop.reindex(index)
pop
```

```
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

Multiple index values

Data

MultiIndex: **a cleaner way**

We can use the MultiIndex to re-index our series
This creates a hierarchical representation of the data

```
index = pd.MultiIndex.from_tuples(index)
index
```

```
MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]]
           labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

```
pop = pop.reindex(index)
pop
```

```
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

Advantage: we can easily access data!
Easier syntax
More efficient operation

```
pop[:, 2010]
```

```
California    37253956
New York      19378102
Texas         25145561
dtype: int64
```

MultiIndex **as extra dimension**

Alternatively, we could have stored the same data using a DataFrame with index and column labels
The unstack() method will convert a multiply indexed Series to a conventionally indexed DataFrame:

```
pop_df = pop.unstack()
pop_df
```

	2000	2010
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

MultiIndex **as extra dimension**

Alternatively, we could have stored the same data using a DataFrame with index and column labels
The unstack() method will convert a multiply indexed Series to a conventionally indexed DataFrame:

```
pop_df = pop.unstack()
pop_df
```

	2000	2010
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

```
pop_df.stack()
```

```
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

.stack() does the opposite operation!

MultiIndex **as extra dimension**

Why would we bother with hierarchical indexing?

- As we were able to use multi-indexing to represent 2D data with 1D Series, we can also use it to represent data of three or more dimensions in a Series or DataFrame
- Each extra level: an extra dimension in the data
- Gives more flexibility in the types of data we can represent
- We may want to add another column of demographic data for each state at each year (e.g. population under 18)
- With MultiIndex it is as easy as adding another column in the DataFrame

```
pop_df = pd.DataFrame({'total': pop,
                       'under18': [9267089, 9284094,
                                   4687374, 4318033,
                                   5906301, 6879014]})
pop_df
```

		total	under18
California	**2000**	33871648	9267089
	2010	37253956	9284094
New York	**2000**	18976457	4687374
	2010	19378102	4318033
Texas	**2000**	20851820	5906301
	2010	25145561	6879014

MultiIndex **as extra dimension**

Another advantage: all the ufuncs and other functionalities work with hierarchical indexes too

Example: we can compute the fraction of people under 18 by year:

```
f_u18 = pop_df['under18'] / pop_df['total']
f_u18.unstack()
```

	2000	2010
California	0.273594	0.249211
New York	0.247010	0.222831
Texas	0.283251	0.273568

How do we create a MultiIndex?

Pass a list of two or more index arrays to a Series or DataFrame:

```
df = pd.DataFrame(np.random.rand(4, 2),
                  index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                  columns=['data1', 'data2'])
df
```

		data1	data2
a	1	0.554233	0.356072
	2	0.925244	0.219474
b	1	0.441759	0.610054
	2	0.171495	0.886688

The indexes of “a”/”b” and “1”/”2” need to be complementary and are matched one on one

The work of creating MultiIndex is done in the background

How do we create a MultiIndex?

Pass a dictionary with appropriate tuples as keys:

```
data = {('California', 2000): 33871648,
        ('California', 2010): 37253956,
        ('Texas', 2000): 20851820,
        ('Texas', 2010): 25145561,
        ('New York', 2000): 18976457,
        ('New York', 2010): 19378102}
pd.Series(data)
```

```
California  2000    33871648
            2010    37253956
Texas       2000    20851820
            2010    25145561
New York    2000    18976457
            2010    19378102
dtype: int64
```

Pandas will automatically recognize the dictionary with tuples as a MultiIndex!

Handling MultiIndex: index names

We can name the indexes of MultiIndex with two ways:
- Passing the names argument to any of the MultiIndex constructors
- Setting the names attribute after their creation:

```
pop.index.names = ['state', 'year']
pop
```

```
state       year
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

It can be a useful way to keep track of the meaning of various index values!

MultiIndex for columns

In DataFrame, the rows and columns are symmetric
Just as the rows can have multiple levels of indices, the columns can have multiple levels too
e.g.

```
# hierarchical indices and columns
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                   names=['year', 'visit'])
columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'], ['HR'
                                     names=['subject', 'type'])

# mock some data
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37

# create the DataFrame
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data
```

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	31.0	38.7	32.0	36.7	35.0	37.2
	2	44.0	37.7	50.0	35.0	29.0	36.7
2014	1	30.0	37.4	39.0	37.8	61.0	36.9
	2	47.0	37.8	48.0	37.3	51.0	36.5

Example where we have multi-indexing
for rows and for columns

MultiIndex for columns

In DataFrame, the rows and columns are symmetric
Just as the rows can have multiple levels of indices, the columns can have multiple levels too
e.g.

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	31.0	38.7	32.0	36.7	35.0	37.2
	2	44.0	37.7	50.0	35.0	29.0	36.7
2014	1	30.0	37.4	39.0	37.8	61.0	36.9
	2	47.0	37.8	48.0	37.3	51.0	36.5

```
health_data['Guido']
```

	type	HR	Temp
year	visit		
2013	1	32.0	36.7
	2	50.0	35.0
2014	1	39.0	37.8
	2	48.0	37.3

Essentially, 4D data
Dimensions:
- Subject
- Measurement type
- Year
- Visit number

If we index the top-level column by the person's name, we can get a full DataFrame containing just this person's information

Today

1. Pandas: recap
2. Indexing part 2 (multi-indexing)
3. **Working with missing data**
4. Concatenating datasets

A very common problem with real world data: Missing Values!

- Missing values are a common problem in many datasets

- Ideas why?

A very common problem with real world data: Missing Values!

- Missing values are a common problem in many datasets

- E.g. a patient withdrew from a study
- A measuring device was out of range
- A system was offline and values are corrupted for a given day

- In a questionnaire some questions were skipped, or answered as “I prefer to not answer”

Missing values: null, NaN NA

- Two approaches to indicate missing values:
  - Using a mask that globally indicates missing values

    or

  - Choosing a sentinel value that indicates a missing entry

Missing values: null, NaN NA

- Two approaches to indicate missing values:
  - Using a mask that globally indicates missing values

A separate Boolean array

Appropriation of 1 bit in the data to indicate null

or

  - Choosing a sentinel value that indicates a missing entry

Data-specific convention e.g. -9999

More general convention e.g. NaN

Missing values in Pandas

- Pandas relies on NumPy → built-in notion of NA values for floating-point data types
- For int-types there is no specific pattern to indicate missing value status
- Pandas uses sentinels for missing data:
  - `NaN` (special floating-point)
  - `None` (Python object)

The Python way to handle missing data: “None”

- The first sentinel value used by Pandas (and Python code in general): **None**
- **None can only be used with arrays of data type = ‘object’**

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

```
vals1 = np.array([1, None, 3, 4])
vals1
```

```
array([1, None, 3, 4], dtype=object)
```

Best common type interpretation that NumPy could infer

The 1st downside of having “None” and data types objects

- Any operations done on the data will be done at Python level with much more overhead than typically fast operations done for arrays with native types:

```
for dtype in ['object', 'int']:
    print("dtype =", dtype)
    %timeit np.arange(1E6, dtype=dtype).sum()
    print()
```

```
dtype = object
10 loops, best of 3: 78.2 ms per loop

dtype = int
100 loops, best of 3: 3.06 ms per loop
```

The 2nd downside of having “None” and data types objects

- Aggregated operations, e.g. sum() or min() in an array with a **None** value will yield an error

- E.g. addition between an integer and None cannot be defined:

```
vals1.sum()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent cal
<ipython-input-4-749fd8ae6030> in <module>()
----> 1 vals1.sum()

/Users/jakevdp/anaconda/lib/python3.5/site-packages/numpy/core/_meth
     30 
     31 def _sum(a, axis=None, dtype=None, out=None, keepdims=False)
---> 32     return umr_sum(a, axis, dtype, out, keepdims)
     33 
     34 def _prod(a, axis=None, dtype=None, out=None, keepdims=False

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

The IEEE way to handle missing data: “NaN: Not a Number”

- **NaN**: a special floating point value recognized by all systems that use the **standard IEEE floating point representation**

- Because NaN is recognized as a **floating-point type**, python will support fast operations, e.g. sum() min() etc

```
vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype
```

```
dtype('float64')
```

Caution with NaN: It infects all other objects that it touches

- Regardless of the operation, the result of arithmetic with NaN will be another NaN

```
1 + np.nan
```

```
nan
```

```
0 *  np.nan
```

```
nan
```

- Aggregates over the values will not give an error, but can contain NaNs:

```
vals2.sum(), vals2.min(), vals2.max()
```

```
(nan, nan, nan)
```

NaN and None in Pandas!

Pandas handles NaN and None interchangeably and converts them when needed:

```
pd.Series([1, np.nan, 2, None])
```

```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

NaN and None in Pandas!

When None or NaN values are present, Pandas will automatically type-cast

If an array is set to integer and we assign to it a NaN value, it will be upcast to a floating-type to accommodate the NaN:

```
x = pd.Series(range(2), dtype=int)
x
```

```
0    0
1    1
dtype: int64
```

```
x[0] = None
x
```

```
0    NaN
1    1.0
dtype: float64
```

None is automatically converted to NaN

Typeclass Conversion when storing NaNs

Typeclass	Conversion When Storing NAs	NA Sentinel Value
`floating`	No change	`np.nan`
`object`	No change	`None` or `np.nan`
`integer`	Cast to `float64`	`np.nan`
`boolean`	Cast to `object`	`None` or `np.nan`

How do we deal with Null values?

- Pandas (and Python) can handle Null values. How do we deal with them?
- Several methods to detect, remove and replace null values in Pandas:

- **isnull()** : generates a Boolean mask to indicate missing values
- **notnull()**: does the opposite of isnull()
- **dropna()**: returns a filled version of the data
- **fillna()**: returns a copy of the data with missing values filled

How do we deal with Null values?

- Pandas (and Python) can handle Null values. How do we deal with them?
- Several methods to detect, remove and replace null values in Pandas:

- **isnull()** : generates a Boolean mask to indicate missing values
- **notnull()**: does the opposite of isnull()

Detecting

- **dropna()**: returns a filled version of the data

Dropping

- **fillna()**: returns a copy of the data with missing values filled

Filling

A. Detecting null values

- **isnull()** : generates a Boolean mask to indicate missing values

- **notnull()**: does the opposite of isnull()

```
data = pd.Series([1, np.nan, 'hello', None])
```

```
data.isnull()
```

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

Boolean masks

A. Detecting null values

- **isnull()** : generates a Boolean mask to indicate missing values
- **notnull()**: does the opposite of isnull()

```
data = pd.Series([1, np.nan, 'hello', None])
```

```
data.isnull()
```

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

Boolean masks
Used as Series or
DataFrame index

```
data[data.notnull()]
```

```
0        1
2    hello
dtype: object
```

B. Dropping null values

- **dropna()**: returns a filled version of the data

```
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])
df
```

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

```
df.dropna()
```

	0	1	2
1	2.0	3.0	5

We cannot drop single values

We drop full rows or full columns

Default: dropping all **rows** with *any* null value

B. Dropping null values

- **dropna()**: returns a filled version of the data

```
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])
df
```

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

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

	2
0	2
1	5
2	6

We cannot drop single values

We drop full rows or full columns

Default: dropping all **rows** with *any* null value

or, we can specify if we want to drop Nulls in another axis

B. Dropping null values

- **dropna()**: **how** and **thres** parameters

```
df[3] = np.nan
df
```

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

```
df.dropna(axis='columns', how='all')
```

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

Only columns where every entry is NaN are dropped with how = ‘all’

Sometimes we don’t want to drop each and every column or row with null

This may result in loss of some good data too

B. Dropping null values

- **dropna()**: **how** and **thres** parameters

```
df[3] = np.nan
df
```

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

```
df.dropna(axis='rows', thresh=3)
```

	0	1	2	3
1	2.0	3.0	5	NaN

Or, we can specify a minimum number of non-null values for the row/column to be kept

C. Filling null values

- **fillna()**: returns a copy of the data with missing values filled

```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data
```

```
a    1.0
b    NaN
c    2.0
d    NaN
e    3.0
dtype: float64
```

Sometimes we want to replace Null values with a valid value, e.g. 0 or interpolated value.

C. Filling null values

- **fillna()**: returns a copy of the data with missing values filled

```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data
```

```
a    1.0
b    NaN
c    2.0
d    NaN
e    3.0
dtype: float64
```

```
data.fillna(0)
```

```
a    1.0
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

We fill NaNs with 0

C. Filling null values

- **fillna()**: returns a copy of the data with missing values filled

```
# forward-fill
data.fillna(method='ffill')
```

```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

We forward-fill to propagate the previous value forward

```
# back-fill
data.fillna(method='bfill')
```

```
a    1.0
b    2.0
c    2.0
d    3.0
e    3.0
dtype: float64
```

We back-fill to propagate the next values backward

C. Filling null values in DataFrames

- **fillna()**: returns a copy of the data with missing values filled

```
df
```

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

```
df.fillna(method='ffill', axis=1)
```

	0	1	2	3
0	1.0	1.0	2.0	2.0
1	2.0	3.0	5.0	5.0
2	NaN	4.0	6.0	6.0

Similar options but we can also specify an axis for the fill

Today

1. Pandas: recap

2. Indexing part 2 (multi-indexing)

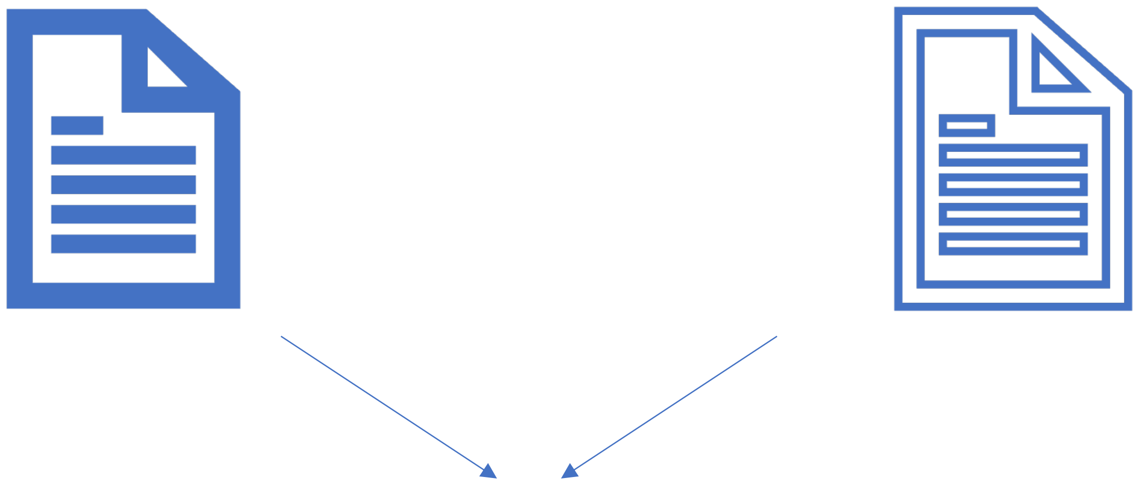
3. Working with missing data

4. **Concatenating datasets**

How do we merge datasets? Examples

Patient records from 2018

Patient records from 2019



Merge

Concatenations: reminder from NumPy

```
x = [1, 2, 3]
y = [4, 5, 6]
z = [7, 8, 9]
np.concatenate([x, y, z])
```

List or tuple of arrays to concatenate

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
x = [[1, 2],
     [3, 4]]
np.concatenate([x, x], axis=1)
```

Axis: specify the axis where results will be concatenated

```
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
```

Concatenations in Pandas

Similar options to NumPy

Can concatenate a DataFrame OR Series

```
pd.concat(objs, axis=0, join='outer', ignore_index=False,
          keys=None, levels=None, names=None,
          verify_integrity=False, sort=False, copy=True)
```

Concatenations in Pandas

Simple case: we concatenate two Series in a similar way as we would concatenate two arrays

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
```

```
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

Concatenations in Pandas

Additionally: can concatenate higher-dimensional objects, like **DataFrames**:
Default concatenation takes place row-wise:

```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
display('df1', 'df2', 'pd.concat([df1, df2])')
```

df1

	A	B
1	A1	B1
2	A2	B2

df2

	A	B
3	A3	B3
4	A4	B4

pd.concat([df1, df2])

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

```
def make_df(cols, ind):
    """Quickly make a DataFrame"""
    data = {c: [str(c) + str(i) for i in ind]
            for c in cols}
    return pd.DataFrame(data, ind)

# example DataFrame
make_df('ABC', range(3))
```

	A	B	C
0	A0	B0	C0
1	A1	B1	C1
2	A2	B2	C2

Function that creates a dataframe

Concatenations in Pandas

Additionally: can concatenate higher-dimensional objects, like **DataFrames**:
Default concatenation takes place row-wise
We can also specify a given axis:

```
df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
display('df3', 'df4', "pd.concat([df3, df4], axis='col')")
```

df3

	A	B
0	A0	B0
1	A1	B1

df4

	C	D
0	C0	D0
1	C1	D1

pd.concat([df3, df4], axis='col')

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1

Duplicate indices during concatenation

Difference between `np.concatenate()` vs. `pd.concat()`

Pandas preserves indices even if the result has duplicate indices:

```
x = make_df('AB', [0, 1])
y = make_df('AB', [2, 3])
y.index = x.index  # make duplicate indices!
display('x', 'y', 'pd.concat([x, y])')
```

x

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

pd.concat([x, y])

	A	B
0	A0	B0
1	A1	B1
0	A2	B2
1	A3	B3

The output has repeated indices!

Undesirable outcome…

Duplicate indices during concatenation

Difference between `np.concatenate()` vs. `pd.concat()`

Pandas preserves indices even if the result has duplicate indices:

```
try:
    pd.concat([x, y], verify_integrity=True)
except ValueError as e:
    print("ValueError:", e)
```

```
ValueError: Indexes have overlapping values: [0, 1]
```

The output has repeated indices!

Undesirable outcome…

We can catch this with an error!

Duplicate indices during concatenation

Difference between `np.concatenate()` vs. `pd.concat()`

Pandas preserves indices even if the result has duplicate indices:

```
display('x', 'y', 'pd.concat([x, y], ignore_index=True)')
```

x

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

pd.concat([x, y], ignore_index=True)

	A	B
0	A0	B0
1	A1	B1
2	A2	B2
3	A3	B3

Indexes are re-organized

The output has repeated indices!

Undesirable outcome…

We can catch this with an error!

Or we can fix it with the flag `ignore_index`

Duplicate indices during concatenation

Difference between `np.concatenate()` vs. `pd.concat()`

We can also use keys to create an hierarchy of indexes, preserving the labels of the data sources:

```
display('x', 'y', "pd.concat([x, y], keys=['x', 'y'])")
```

x

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

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

		A	B
x	0	A0	B0
	1	A1	B1
y	0	A2	B2
	1	A3	B3

MultiIndexed DataFrames

The output has repeated indices!

Undesirable outcome…

We can catch this with an error!

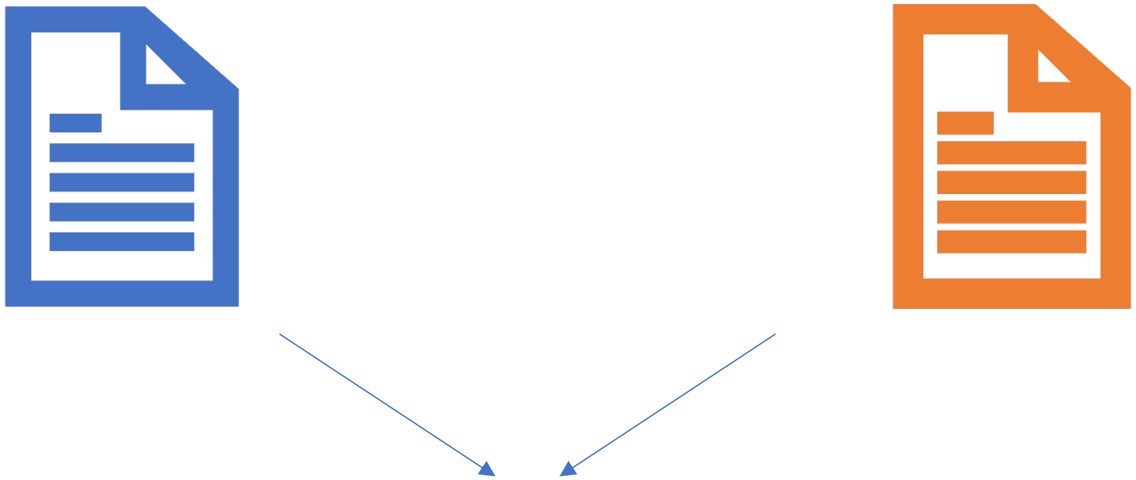
Or we can add hierarchical indexes

What happens when data have different sets of column names?

What if we now want to merge data from 2 different hospitals, that do not share all information?

Patient records from Hospital A

Patient records from Hospital B



Merge?

What happens when data have different sets of column names?

Example:

```
df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
display('df5', 'df6', 'pd.concat([df5, df6])')
```

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

pd.concat([df5, df6])

	A	B	C	D
1	A1	B1	C1	NaN
2	A2	B2	C2	NaN
3	NaN	B3	C3	D3
4	NaN	B4	C4	D4

What happens when data have different sets of column names?

Example:

```
df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
display('df5', 'df6', 'pd.concat([df5, df6])')
```

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

pd.concat([df5, df6])

	A	B	C	D
1	A1	B1	C1	NaN
2	A2	B2	C2	NaN
3	NaN	B3	C3	D3
4	NaN	B4	C4	D4

Default: several missing values

What happens when data have different sets of column names?

We have other options for ”join” while concatenating the two sets:

```
display('df5', 'df6',
        "pd.concat([df5, df6], join='inner')")
```

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

pd.concat([df5, df6], join='inner')

	B	C
1	B1	C1
2	B2	C2
3	B3	C3
4	B4	C4

`join = ‘inner’` will keep the intersection of columns

Appending in pandas

Appending is similar to concatenate
Instead of `pd.concat([df1, df2])` you can also call *`df1.append(df2):`*

```
display('df1', 'df2', 'df1.append(df2)')
```

df1

	A	B
1	A1	B1
2	A2	B2

df2

	A	B
3	A3	B3
4	A4	B4

df1.append(df2)

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

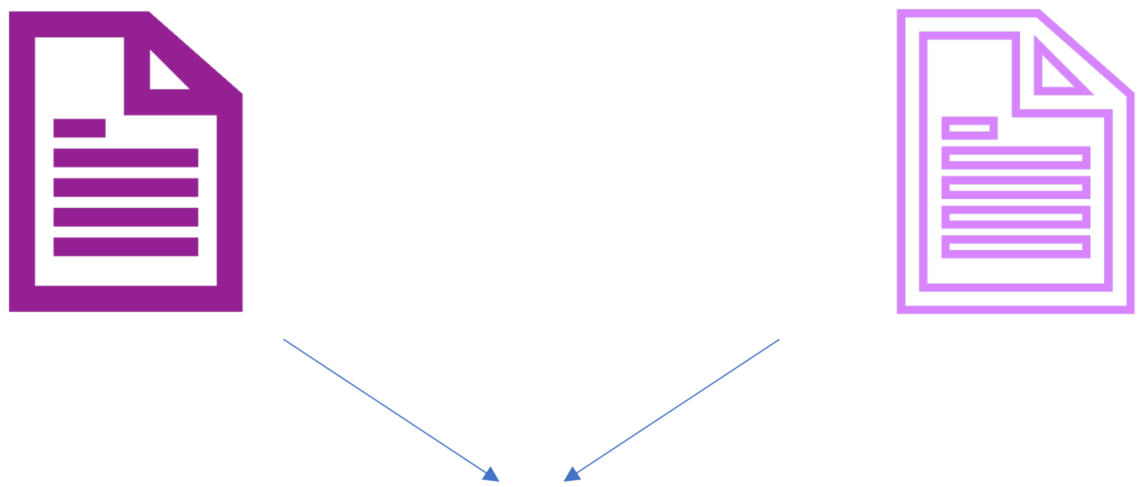
It does not modify the original object (unlike lists); it creates a new object with the combined data (not very efficient)

What happens when different parts of our data are stored in different DataFrames?

Now, we want to combine two datasets, e.g. one with names and function;
another with names and salaries

Staff names and function

Staff names and salaries



Merge and
Join!

One – to – One joins

Similar to a column-wise concatenation

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineer
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
display('df1', 'df2')
```

Two DataFrames: have different information on the same employees

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

```
df3 = pd.merge(df1, df2)
df3
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

We merge them to combine this info into one single DataFrame

Column 'Employee' exists in both DataFrames and is used to join them

Many – to – One joins

One of the two key columns contain duplicate entries

```python
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
display('df3', 'df4', 'pd.merge(df3, df4)')
```

df3

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

df4

	group	supervisor
0	Accounting	Carly
1	Engineering	Guido
2	HR	Steve

One of the key columns where the merge is based on has duplicate entries

pd.merge(df3, df4)

	employee	group	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3	Sue	HR	2014	Steve

The result will have an additional column with “Supervisor” and will repeat the duplicate information

Many – to – Many joins

The key column in *both the left and right array contains duplicates*

```
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                              'Engineering', 'Engineering', 'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'coding', 'linux',
                               'spreadsheets', 'organization']})
display('df1', 'df5', "pd.merge(df1, df5)")
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df5

	group	skills
0	Accounting	math
1	Accounting	spreadsheets
2	Engineering	coding
3	Engineering	linux
4	HR	spreadsheets
5	HR	organization

Both of the key columns contain duplicates!

Can you find the duplicate?

What will happen?

Many – to – Many joins

```python
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                              'Engineering', 'Engineering', 'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'coding', 'linux',
                               'spreadsheets', 'organization']})
display('df1', 'df5', "pd.merge(df1, df5)")
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df5

	group	skills
0	Accounting	math
1	Accounting	spreadsheets
2	Engineering	coding
3	Engineering	linux
4	HR	spreadsheets
5	HR	organization

pd.merge(df1, df5)

	employee	group	skills
0	Bob	Accounting	math
1	Bob	Accounting	spreadsheets
2	Jake	Engineering	coding
3	Jake	Engineering	linux
4	Lisa	Engineering	coding
5	Lisa	Engineering	linux
6	Sue	HR	spreadsheets
7	Sue	HR	organization

Both of the key columns contain duplicates!

Every combination of duplicate columns is repeated in the output of "merge":

Can we specify the Merge Key?

We can explicitly define which column we want to merge on with the “on” parameter

```
display('df1', 'df2', "pd.merge(df1, df2, on='employee')")
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

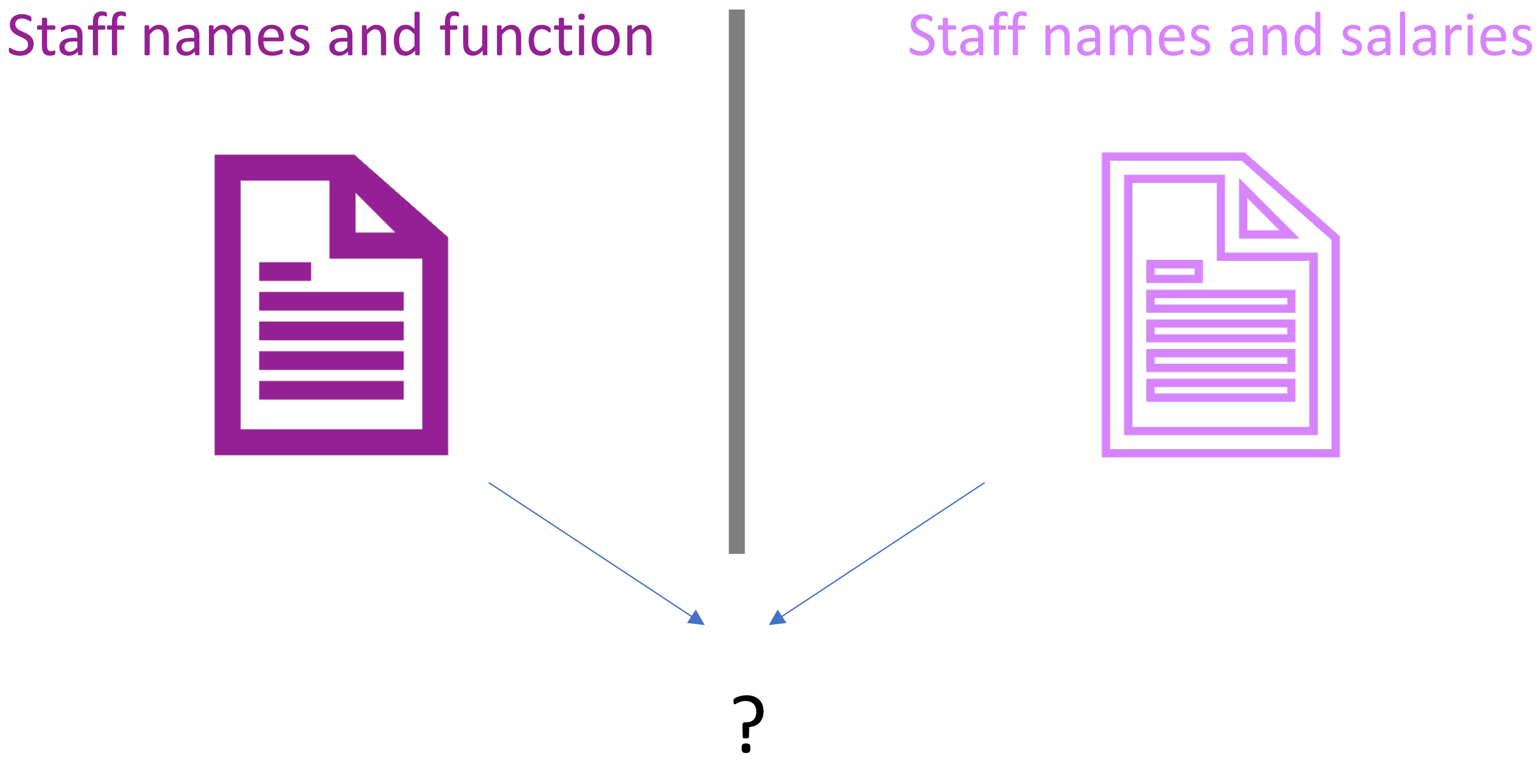
df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

pd.merge(df1, df2, on='employee')

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

What happens when different parts of our data are stored in different DataFrames?



Example: we want to combine two datasets, e.g. one with names and function; another with names and salaries

Our two services did not discuss with each other and now the keys are different. What happens?

Can we specify the Merge Key when keys do not match?

We may want to merge datasets with different column names
For example “name” or “employee”

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df3

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

```
pd.merge(df1, df3, left_on="employee",
right_on="name")
```

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

“left_on”
“right_on”

Specify the two column names that we want to use for merging

Can we specify the Merge Key when keys do not match?

We may want to merge datasets with different column names
For example “name” or “employee”
We can also drop redundant columns:

```
pd.merge(df1, df3, left_on="employee",
right_on="name").drop('name', axis=1)
```

	employee	group	salary
0	Bob	Accounting	70000
1	Jake	Engineering	80000
2	Lisa	Engineering	120000
3	Sue	HR	90000

Merging on an index

Rather than merging on a column, we can also merge on an index.

```
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
```

```
df1a
```

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

```
df2a
```

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

```
pd.merge(df1a, df2a, left_index=True,
right_index=True)
```

	group	hire_date
employee		
Lisa	Engineering	2004
Bob	Accounting	2008
Jake	Engineering	2012
Sue	HR	2014

Merging on an index

Rather than merging on a column, we can also merge on an index.
The .join() method will perform a merge that by default joins indexes:

```
display('df1a', 'df2a', 'df1a.join(df2a)')
```

df1a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

df1a.join(df2a)

	group	hire_date
employee		
Bob	Accounting	2008
Jake	Engineering	2012
Lisa	Engineering	2004
Sue	HR	2014

Today

1. Pandas: recap
2. Indexing part 2 (multi-indexing)
3. Working with missing data
4. Concatenating datasets