Data Processing and Visualizations using Python

Day 5 – pandas (Part 2)

SICSS 2022 – Haifa University

Amit Donner

- Today we will dive deeper into pandas and introduce additional important tools.
- For example, dealing with missing values, duplicates, transformations, categorical variables and dummy variables.
- We will discuss the differences between long and wide formats.
- Moreover, we will learn how to efficiently merge datasets and distinguish between the different options.
- Finally, we will implement our new knowledge and perform a small analysis on the Covid 19 data from Israel.
- Before all of that, the first step is always:

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

- For the following example we will use this ‘toy’ data

```python
dat = pd.DataFrame({"A": [1, 2, 3, 1],
                    "B": [10, np.nan, -4, 10],
                    "C": [3.2, 2, np.nan, 3.2],
                    "D": [1, 0, 3, 1]})
dat
```

	A	B	C	D
0	1	10.0	3.2	1
1	2	NaN	2.0	0
2	3	-4.0	NaN	3
3	1	10.0	3.2	1

- Observe the presence of missing values, represented by “NaN”, and that we have 2 identical observations.

DataFrame.transform()

- Assume we wish to compute the exponent (i.e. e^x) of the first and second columns in the data. The transform method enables us do so with ease.

```
dat[["A","B"]].transform(np.exp)
```

	A	B
0	2.718282	22026.465795
1	7.389056	NaN
2	20.085537	0.018316
3	2.718282	22026.465795

- Observe that when computing the exponent of the NaN value, it remains NaN.

- Note that in order to change the values in the data itself we should use assignment.

```
dat[["A","B"]] = dat[["A","B"]].transform(np.exp)
dat
```

	A	B	C	D
0	2.718282	22026.465795	3.2	1
1	7.389056	NaN	2.0	0
2	20.085537	0.018316	NaN	3
3	2.718282	22026.465795	3.2	1

- Another option is to use a self-defined function, for example:

```
def f(x):
    return np.abs((x+5)*2)
dat[["A","B"]].transform(f)
```

	A	B
0	12.0	30.0
1	14.0	NaN
2	16.0	2.0
3	12.0	30.0

Lambda functions

- Sometimes, we wish to apply a short computation that has no specific function (e.g., computing $|(x+5)\cdot 2|$).
- Instead of defining a function before hand, we can use a *lambda* function.
- In simple words – a short function that is defined in one row inside the transformation method.
- If we use *lambda* function in the example from last slide, it will look like this:

```
dat[["A","B"]].transform(lambda x: np.abs((x+5)*2))
```

	A	B
0	12.0	30.0
1	14.0	NaN
2	16.0	2.0
3	12.0	30.0

- What about creating a new variable which is based on others?
- Using the *assign* method, we can compute new variables which are result of operation on other variables in the data.

```
dat.assign(E = 2*dat.D/dat.C)
```

	A	B	C	D	E
0	1.0	10.0	3.2	1	0.625
1	2.0	NaN	2.0	0	0.000
2	3.0	-4.0	NaN	3	NaN
3	1.0	10.0	3.2	1	0.625

```
dat
```

	A	B	C	D
0	1.0	10.0	3.2	1
1	2.0	NaN	2.0	0
2	3.0	-4.0	NaN	3
3	1.0	10.0	3.2	1

- In order to "keep" the new variable in the data set we once again need to do an assignment.

```
dat = dat.assign(E = 2*dat.D/dat.C)
dat
```

	A	B	C	D	E
0	1.0	10.0	3.2	1	0.625
1	2.0	NaN	2.0	0	0.000
2	3.0	-4.0	NaN	3	NaN
3	1.0	10.0	3.2	1	0.625

- Another option, is to just assign a new variable “E”.

```
dat["E"] = 2*dat.D/dat.C
dat
```

	A	B	C	D	E
0	1.0	10.0	3.2	1	0.625
1	2.0	NaN	2.0	0	0.000
2	3.0	-4.0	NaN	3	NaN
3	1.0	10.0	3.2	1	0.625

- Observe that the NaN value remained after all the computations we did.

Missing values

- Missing, out of range values (e.g., negative age, number instead of a name, proportion higher than 1 and so on).
- Extreme values, outliers…
- Only a few examples of anomalies we encounter frequently when working with observational data.
- Many approaches have been suggested for dealing with missing values such as to omit them, imputations, and so on…
- In our course we will demonstrate only basic methods for dealing with such cases.

- First, we need to be able to detect missing values. The *isna()* method goes over *every* value in the data and returns an equal dimensions array with *True* if the corresponding value is missing, or *False* otherwise.

```
dat.isna()
```

	A	B	C	D	E
0	False	False	False	False	False
1	False	True	False	False	False
2	False	False	True	False	True
3	False	False	False	False	False

- Obviously, most data sets are much larger, so we need an option to detect missing values' existence without being needing to go over all the data. The *any()* and *all()* methods are exactly what we need.

```
dat.isna().any()
```

```
A    False
B     True
C     True
D    False
E     True
dtype: bool
```

```
dat.isna().all()
```

```
A    False
B    False
C    False
D    False
E    False
dtype: bool
```

- What about missing values inflated variables? We can compute the proportion of missing values.

```
dat.isna().mean()
```

```
A    0.00
B    0.25
C    0.25
D    0.00
E    0.25
dtype: float64
```

- We can also omit variables with a proportion of missing values that is higher than some threshold.

```
dat.loc[:,dat.isna().mean() < 0.1]
```

	A	D
0	1.0	1
1	2.0	0
2	3.0	3
3	1.0	1

DataFrame.fillna()

- The *fillna* method is useful for replacing missing values with some other constant (e.g., 0, the column's mean, median and others).

```
dat.fillna(0)
```

	A	B	C	D	E
0	1	10.0	3.2	1	0.625
1	2	0.0	2.0	0	0.000
2	3	-4.0	0.0	3	0.000
3	1	10.0	3.2	1	0.625

```
dat.fillna(dat.mean())
```

	A	B	C	D	E
0	1	10.000000	3.2	1	0.625000
1	2	5.333333	2.0	0	0.000000
2	3	-4.000000	2.8	3	0.416667
3	1	10.000000	3.2	1	0.625000

```
dat.fillna(dat.median())
```

	A	B	C	D	E
0	1.0	10.0	3.2	1	0.625
1	2.0	10.0	2.0	0	0.000
2	3.0	-4.0	3.2	3	0.625
3	1.0	10.0	3.2	1	0.625

- Again, this method don’t replace that values in the data.

```
dat
```

	A	B	C	D	E
0	1	10.0	3.2	1	0.625
1	2	NaN	2.0	0	0.000
2	3	-4.0	NaN	3	NaN
3	1	10.0	3.2	1	0.625

- The *fillna* (and some other) methods include the *inplace* argument which enables us to change the data frame’s values and not just change them temporarily.

```
dat.fillna(dat.mean(), inplace=True)
dat
```

	A	B	C	D	E
0	1.0	10.000000	3.2	1	0.625000
1	2.0	5.333333	2.0	0	0.000000
2	3.0	-4.000000	2.8	3	0.416667
3	1.0	10.000000	3.2	1	0.625000

- An additional option is to omit rows with at least one missing value using the *dropna* method (Use with caution).

```
dat
```

	A	B	C	D
0	1	10.0	3.2	1
1	2	NaN	2.0	0
2	3	-4.0	NaN	3
3	1	10.0	3.2	1

```
dat.dropna()
```

	A	B	C	D
0	1	10.0	3.2	1
3	1	10.0	3.2	1

- The last remark will be about duplicates. Observe that the first and last observations are identical. Sometimes observation are included more than once by mistake, and we need to omit the duplicates.

```
dat.drop_duplicates()
```

	A	B	C	D	E
0	1	10.000000	3.2	1	0.625000
1	2	5.333333	2.0	0	0.000000
2	3	-4.000000	2.8	3	0.416667

DataFrame.get_dummies()

- An additional important method, that converts categorical variables to zero-one dummies.
- Consider the following toy data:

```
dat1 = pd.DataFrame({"Car": ["Toyota", "Toyota", "Suzuki", "Toyota", "Suzuki", "Mazda", "Mazda"],
                     "Gear": ["Manual", "Manual", "Manual", "Automatic", "Automatic", "Automatic", "Automatic"],
                     "Price": [100, 102, 80, 110, 150, 132, 140]})
dat1
```

	Car	Gear	Price
0	Toyota	Manual	100
1	Toyota	Manual	102
2	Suzuki	Manual	80
3	Toyota	Automatic	110
4	Suzuki	Automatic	150
5	Mazda	Automatic	132
6	Mazda	Automatic	140

- First, we will convert the *car* column to dummies.

```
pd.get_dummies(dat1, columns=['Car'])
```

	Gear	Price	Car_Mazda	Car_Suzuki	Car_Toyota
0	Manual	100	0	0	1
1	Manual	102	0	0	1
2	Manual	80	0	1	0
3	Automatic	110	0	0	1
4	Automatic	150	0	1	0
5	Automatic	132	1	0	0
6	Automatic	140	1	0	0

- Usually, for categorical variable with k levels we need only $k-1$ dummies, this can be done also by passing *drop_first = True.*

```
pd.get_dummies(dat1, columns=['Car'], drop_first=True)
```

	Gear	Price	Car_Suzuki	Car_Toyota
0	Manual	100	0	1
1	Manual	102	0	1
2	Manual	80	1	0
3	Automatic	110	0	1
4	Automatic	150	1	0
5	Automatic	132	0	0
6	Automatic	140	0	0

- We can also pass more than one variable when using this method.

```
pd.get_dummies(dat1, columns=['Car', 'Gear'])
```

	Price	Car_Mazda	Car_Suzuki	Car_Toyota	Gear_Automatic	Gear_Manual
0	100	0	0	1	0	1
1	102	0	0	1	0	1
2	80	0	1	0	0	1
3	110	0	0	1	1	0
4	150	0	1	0	1	0
5	132	1	0	0	1	0
6	140	1	0	0	1	0

- The *drop_first* argument works in this case as well.

```
pd.get_dummies(dat1, columns=['Car', 'Gear'], drop_first=True)
```

	Price	Car_Suzuki	Car_Toyota	Gear_Manual
0	100	0	1	1
1	102	0	1	1
2	80	1	0	1
3	110	0	1	0
4	150	1	0	0
5	132	0	0	0
6	140	0	0	0

Wide and long formats

- Consider an experiment where each subject has more than 1 measurement (different time points, pre-post treatment, etc..).
- How to store the results in a dataset?
- Wide format – new variable for each measurement and one observation (in the simple cases) for each participant.
- Long format – One variable for all the measurements, another one that indicates the measurement index (i.e., 1,2,…) and a third one which distinguishes between participants.

- Consider the following data and observe that it is in wide format.

	Participant	Measure 1	Measure 2	Measure 3
0	1	102.3	90	105.0
1	2	90.0	88	98.0
2	3	75.0	91	103.2
3	4	69.0	70	100.0

- Its long format equivalent is:

	Participant	Measure	Value
0	1	Measure 1	102.3
1	2	Measure 1	90.0
2	3	Measure 1	75.0
3	4	Measure 1	69.0
4	1	Measure 2	90.0
5	2	Measure 2	88.0
6	3	Measure 2	91.0
7	4	Measure 2	70.0
8	1	Measure 3	105.0
9	2	Measure 3	98.0
10	3	Measure 3	103.2
11	4	Measure 3	100.0

Which one is better?

- In most cases, long format are more suitable for statistical and machine learning models and for visualizations.

From wide to long

- The *melt* function lets us convert from wide to long.
- First attempt:

```
pd.melt(dat_wide)
```

	Participant	Measure 1	Measure 2	Measure 3
0	1	102.3	90	105.0
1	2	90.0	88	98.0
2	3	75.0	91	103.2
3	4	69.0	70	100.0

	variable	value
0	Participant	1.0
1	Participant	2.0
2	Participant	3.0
3	Participant	4.0
4	Measure 1	102.3
5	Measure 1	90.0
6	Measure 1	75.0
7	Measure 1	69.0
8	Measure 2	90.0
9	Measure 2	88.0
10	Measure 2	91.0
11	Measure 2	70.0
12	Measure 3	105.0
13	Measure 3	98.0
14	Measure 3	103.2
15	Measure 3	100.0

- In order to perform it correctly, we should specify the variables that we don't want to pivot.

```
pd.melt(dat_wide, id_vars=['Participant'])
```

	Participant	variable	value
0	1	Measure 1	102.3
1	2	Measure 1	90.0
2	3	Measure 1	75.0
3	4	Measure 1	69.0
4	1	Measure 2	90.0
5	2	Measure 2	88.0
6	3	Measure 2	91.0
7	4	Measure 2	70.0
8	1	Measure 3	105.0
9	2	Measure 3	98.0
10	3	Measure 3	103.2
11	4	Measure 3	100.0

- Using the *var_name* and *value_name* arguments we can control the new variable names.

```
dat_long = pd.melt(dat_wide,
        var_name = "Measure",
        id_vars= 'Participant',
        value_name= 'Value' )
dat_long
```

	Participant	Measure	Value
0	1	Measure 1	102.3
1	2	Measure 1	90.0
2	3	Measure 1	75.0
3	4	Measure 1	69.0
4	1	Measure 2	90.0
5	2	Measure 2	88.0
6	3	Measure 2	91.0
7	4	Measure 2	70.0
8	1	Measure 3	105.0
9	2	Measure 3	98.0
10	3	Measure 3	103.2
11	4	Measure 3	100.0

From long to wide

- What about the opposite direction? Easy-peasy!
- Using the *pivot* function, we can perform this procedure.

```
pd.pivot(data = dat_long, index = 'Participant', columns = 'Measure', values = 'Value')
```

Measure	Measure 1	Measure 2	Measure 3
Participant			
1	102.3	90.0	105.0
2	90.0	88.0	98.0
3	75.0	91.0	103.2
4	69.0	70.0	100.0

- Observe that we need to pass 3 arguments:
- *Index* – Which columns that should be the new index.
- *Columns* – Which variable should determine the new columns.
- *Values* – Where to take the values from.

- Observe that the resulted data frame is not exactly the wide data we started with, because now the participant variable is the data frame index, and not regular column. In order to fix that, we can do the following:

```
pd.pivot(data = dat_long, index = 'Participant', columns = 'Measure', values = 'Value').reset_index()
```

Measure	Participant	Measure 1	Measure 2	Measure 3
0	1	102.3	90.0	105.0
1	2	90.0	88.0	98.0
2	3	75.0	91.0	103.2
3	4	69.0	70.0	100.0

```
pd.pivot(data = dat_long, index = 'Participant', columns = 'Measure', values = 'Value').reset_index()\
.rename_axis(None, axis = 1)
```

	Participant	Measure 1	Measure 2	Measure 3
0	1	102.3	90.0	105.0
1	2	90.0	88.0	98.0
2	3	75.0	91.0	103.2
3	4	69.0	70.0	100.0

- Now, assume that each participant had 4 different sessions, where in each one he was measured 3 times.

```
dat_wide
```

	Participant	Session	Measure 1	Measure 2	Measure 3
0	1	1	102.3	90	105.0
1	1	2	90.0	88	98.0
2	1	3	75.0	91	103.2
3	1	4	69.0	70	100.0
4	2	1	99.0	80	95.0
5	2	2	88.0	68	93.0
6	2	3	95.0	71	100.0
7	2	4	89.0	50	70.0
8	3	1	120.0	85	95.0
9	3	2	100.0	90	98.0
10	3	3	85.0	81	103.2
11	3	4	60.0	75	70.0
12	4	1	105.3	80	100.0
13	4	2	70.0	86	90.0
14	4	3	25.0	71	87.0
15	4	4	10.0	60	88.0

- If we want to change this data to a long format, we can still use the *melt* function but now we need to pass *id_vars = ["Participant", "Session"].*

```
dat_long = pd.melt(dat_wide, id_vars = ['Participant', 'Session'],
                             var_name="Measure",
                             value_name="Value")
dat_long.head(15)
```

	Participant	Session	Measure	Value
0	1	1	Measure 1	102.3
1	1	2	Measure 1	90.0
2	1	3	Measure 1	75.0
3	1	4	Measure 1	69.0
4	2	1	Measure 1	99.0
5	2	2	Measure 1	88.0
6	2	3	Measure 1	95.0
7	2	4	Measure 1	89.0
8	3	1	Measure 1	120.0
9	3	2	Measure 1	100.0
10	3	3	Measure 1	85.0
11	3	4	Measure 1	60.0
12	4	1	Measure 1	105.3
13	4	2	Measure 1	70.0
14	4	3	Measure 1	25.0

- What about the opposite direction? Just use *pivot* with *index = [‘Participant’, ‘Session’].*

```
pd.pivot(data = dat_long, index = ['Participant', 'Session'],
                              columns ="Measure",
                              values="Value")
```

	Measure	Measure 1	Measure 2	Measure 3
Participant	**Session**			
1	**1**	102.3	90.0	105.0
	2	90.0	88.0	98.0
	3	75.0	91.0	103.2
	4	69.0	70.0	100.0
2	**1**	99.0	80.0	95.0
	2	88.0	68.0	93.0
	3	95.0	71.0	100.0
	4	89.0	50.0	70.0
3	**1**	120.0	85.0	95.0
	2	100.0	90.0	98.0
	3	85.0	81.0	103.2
	4	60.0	75.0	70.0
4	**1**	105.3	80.0	100.0
	2	70.0	86.0	90.0
	3	25.0	71.0	87.0
	4	10.0	60.0	88.0

- And again, we can reset the indices.

```
pd.pivot(data = dat_long, index = ['Participant', 'Session'],
                         columns ="Measure",
                         values="Value").reset_index().rename_axis(None, axis = 1)
```

	Participant	Session	Measure 1	Measure 2	Measure 3
0	1	1	102.3	90.0	105.0
1	1	2	90.0	88.0	98.0
2	1	3	75.0	91.0	103.2
3	1	4	69.0	70.0	100.0
4	2	1	99.0	80.0	95.0
5	2	2	88.0	68.0	93.0
6	2	3	95.0	71.0	100.0
7	2	4	89.0	50.0	70.0
8	3	1	120.0	85.0	95.0
9	3	2	100.0	90.0	98.0
10	3	3	85.0	81.0	103.2
11	3	4	60.0	75.0	70.0
12	4	1	105.3	80.0	100.0
13	4	2	70.0	86.0	90.0
14	4	3	25.0	71.0	87.0
15	4	4	10.0	60.0	88.0

Merging data frames.

- Consider the last wide formatted data frame we had and assume that we have another small data set with participants characteristics.

	Participant	Session	Measure 1	Measure 2	Measure 3
0	1	1	102.3	90.0	105.0
1	1	2	90.0	88.0	98.0
2	1	3	75.0	91.0	103.2
3	1	4	69.0	70.0	100.0
4	2	1	99.0	80.0	95.0
5	2	2	88.0	68.0	93.0
6	2	3	95.0	71.0	100.0
7	2	4	89.0	50.0	70.0
8	3	1	120.0	85.0	95.0
9	3	2	100.0	90.0	98.0
10	3	3	85.0	81.0	103.2
11	3	4	60.0	75.0	70.0
12	4	1	105.3	80.0	100.0
13	4	2	70.0	86.0	90.0
14	4	3	25.0	71.0	87.0
15	4	4	10.0	60.0	88.0

```
Participants
```

	Participant	Height	IQ
0	1	170	110
1	2	167	98
2	3	185	120
3	4	159	85

- Assume we wish to combine these 2 data frames into 1, such that we maintain the structure of the wide data. Using the *merge* method, we can do so.

```
dat_wide.merge(Participants, on = "Participant")
```

	Participant	Session	Measure 1	Measure 2	Measure 3	Height	IQ
0	1	1	102.3	90	105.0	170	110
1	1	2	90.0	88	98.0	170	110
2	1	3	75.0	91	103.2	170	110
3	1	4	69.0	70	100.0	170	110
4	2	1	99.0	80	95.0	167	98
5	2	2	88.0	68	93.0	167	98
6	2	3	95.0	71	100.0	167	98
7	2	4	89.0	50	70.0	167	98
8	3	1	120.0	85	95.0	185	120
9	3	2	100.0	90	98.0	185	120
10	3	3	85.0	81	103.2	185	120
11	3	4	60.0	75	70.0	185	120
12	4	1	105.3	80	100.0	159	85
13	4	2	70.0	86	90.0	159	85
14	4	3	25.0	71	87.0	159	85
15	4	4	10.0	60	88.0	159	85

- Observe that we used the *on* argument, that determines the variable we merge by.

Left, Right, inner, outer.

- Consider the case where we observe characteristics on the first 3 participants and on some other participant with no measures.

```
Participants1 = pd.DataFrame({"Participant": [1, 2, 3, 5],
                              "Height": [170, 167, 185, 159],
                              "IQ": [110, 98, 120, 85]})
Participants1
```

	Participant	Height	IQ
0	1	170	110
1	2	167	98
2	3	185	120
3	5	159	85

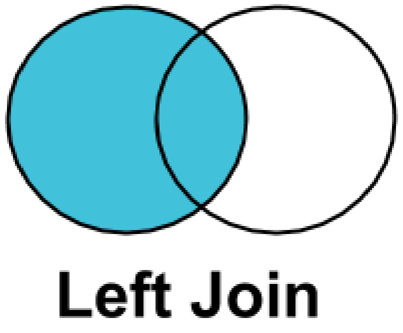
- We can merge the two data sets in different ways.

Left

```
dat_wide.merge(Participants1, how = "left")
```

	Participant	Session	Measure 1	Measure 2	Measure 3	Height	IQ
0	1	1	102.3	90	105.0	170.0	110.0
1	1	2	90.0	88	98.0	170.0	110.0
2	1	3	75.0	91	103.2	170.0	110.0
3	1	4	69.0	70	100.0	170.0	110.0
4	2	1	99.0	80	95.0	167.0	98.0
5	2	2	88.0	68	93.0	167.0	98.0
6	2	3	95.0	71	100.0	167.0	98.0
7	2	4	89.0	50	70.0	167.0	98.0
8	3	1	120.0	85	95.0	185.0	120.0
9	3	2	100.0	90	98.0	185.0	120.0
10	3	3	85.0	81	103.2	185.0	120.0
11	3	4	60.0	75	70.0	185.0	120.0
12	4	1	105.3	80	100.0	NaN	NaN
13	4	2	70.0	86	90.0	NaN	NaN
14	4	3	25.0	71	87.0	NaN	NaN
15	4	4	10.0	60	88.0	NaN	NaN

The participants from the *dat_wide* data set. Observe that it is on the left.

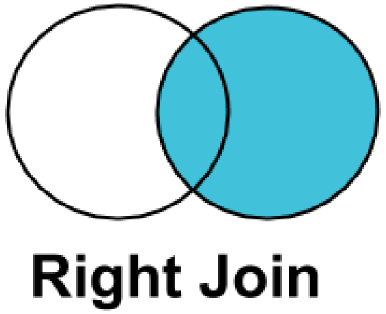


Right

```
dat_wide.merge(Participants1, how = "right")
```

	Participant	Session	Measure 1	Measure 2	Measure 3	Height	IQ
0	1	1.0	102.3	90.0	105.0	170	110
1	1	2.0	90.0	88.0	98.0	170	110
2	1	3.0	75.0	91.0	103.2	170	110
3	1	4.0	69.0	70.0	100.0	170	110
4	2	1.0	99.0	80.0	95.0	167	98
5	2	2.0	88.0	68.0	93.0	167	98
6	2	3.0	95.0	71.0	100.0	167	98
7	2	4.0	89.0	50.0	70.0	167	98
8	3	1.0	120.0	85.0	95.0	185	120
9	3	2.0	100.0	90.0	98.0	185	120
10	3	3.0	85.0	81.0	103.2	185	120
11	3	4.0	60.0	75.0	70.0	185	120
12	5	NaN	NaN	NaN	NaN	159	85

The participants from the *participant1* data set. Observe that it is on the right.

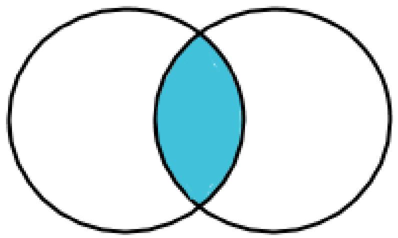


Inner

```
dat_wide.merge(Participants1, how = "inner")
```

	Participant	Session	Measure 1	Measure 2	Measure 3	Height	IQ
0	1	1	102.3	90	105.0	170	110
1	1	2	90.0	88	98.0	170	110
2	1	3	75.0	91	103.2	170	110
3	1	4	69.0	70	100.0	170	110
4	2	1	99.0	80	95.0	167	98
5	2	2	88.0	68	93.0	167	98
6	2	3	95.0	71	100.0	167	98
7	2	4	89.0	50	70.0	167	98
8	3	1	120.0	85	95.0	185	120
9	3	2	100.0	90	98.0	185	120
10	3	3	85.0	81	103.2	185	120
11	3	4	60.0	75	70.0	185	120

The mutual participants in both data sets



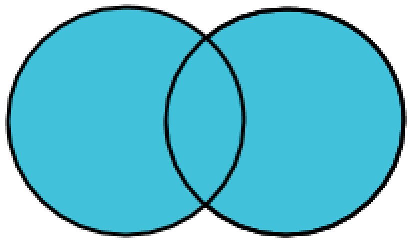
Inner Join

Outer

```
dat_wide.merge(Participants1, how = "outer")
```

	Participant	Session	Measure 1	Measure 2	Measure 3	Height	IQ
0	1	1.0	102.3	90.0	105.0	170.0	110.0
1	1	2.0	90.0	88.0	98.0	170.0	110.0
2	1	3.0	75.0	91.0	103.2	170.0	110.0
3	1	4.0	69.0	70.0	100.0	170.0	110.0
4	2	1.0	99.0	80.0	95.0	167.0	98.0
5	2	2.0	88.0	68.0	93.0	167.0	98.0
6	2	3.0	95.0	71.0	100.0	167.0	98.0
7	2	4.0	89.0	50.0	70.0	167.0	98.0
8	3	1.0	120.0	85.0	95.0	185.0	120.0
9	3	2.0	100.0	90.0	98.0	185.0	120.0
10	3	3.0	85.0	81.0	103.2	185.0	120.0
11	3	4.0	60.0	75.0	70.0	185.0	120.0
12	4	1.0	105.3	80.0	100.0	NaN	NaN
13	4	2.0	70.0	86.0	90.0	NaN	NaN
14	4	3.0	25.0	71.0	87.0	NaN	NaN
15	4	4.0	10.0	60.0	88.0	NaN	NaN
16	5	NaN	NaN	NaN	NaN	159.0	85.0

All participants from both data sets



Full Outer Join