Download datasets:   
<https://drive.google.com/drive/folders/1mS7IM7kwohmGHuFMol3VL-oqF16b_Tv3?usp=sharing>

Or

<https://drive.google.com/drive/folders/1p0mhctIjBiDKT-UGw9EatscSJdNVihfi?usp=share_link>

**Pandas-02**

**Outline**

* **Fetching data using pandas**
  + Drawing a contrast with SQL
  + Masking
  + Grouping
    - Split, Apply, Combine
    - groupby()
    - reset\_index()
    - Filtering
    - Transformation
    - Apply
* **Dealing with missing values**
  + Finding number of missing vals in a column:
    - [df.info](http://df.info/)()
    - df.isna().sum()
  + Removing null vals
    - replace()
    - backward fill
    - forward fill

import pandas as pd

import numpy as np

**Lets import our dataset once again**

* Link: <https://drive.google.com/file/d/1fOeDQuMaV0Mp5eFHPxlrRX6lgq-23z35/view?usp=sharing>

!wget "https://drive.google.com/uc?export=download&id=1fOeDQuMaV0Mp5eFHPxlrRX6lgq-23z35" -O gapminder.csv

gapminder.csv 100%[===================>] 81.82K --.-KB/s in 0.002s

df = pd.read\_csv('gapminder.csv')

**Fetching data using SQL-like queries**

**Now lets say we want to know what all countries have a life expectancy > 60 in any year**

**How can we do this ?**

* This is where concepts of fetching data using pandas comes into picture
* Its actually very similar to SQL

**Have you ever come across or seen SQL?**

* For now, just know that **SQL is used to fetch data from databases**
* To interact with or **query the databases**
* We can use **Pandas operations** in a similar way to **fetch desired data from loaded data**

**Masking**

**Now How can we filter values the rows with lifeExp > 60?**

* In SQL: SELECT \* FROM df WHERE lifeExp > 60
* In pandas:

df.lifeExp > 60

# We haven't filtered yet

# We have just mapped each value in column to `True` or `False`

# based on lifeExp > 60

0 False

1 False

2 False

3 False

4 False

...

1699 True

1700 True

1701 False

1702 False

1703 False

Name: lifeExp, Length: 1704, dtype: bool

This is known as Masking

But we still don’t know the row values … Only that which row satisfied the condtion

**How do we get the row values then ?**

* By applying df.loc()
* This is known as filtering

# Now this is filtering

df.loc[df.lifeExp > 60]

|  | **country** | **continent** | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- | --- | --- |
| **14** | Albania | Europe | 1962 | 64.820 | 1728137 | 2312.888958 |
| **15** | Albania | Europe | 1967 | 66.220 | 1984060 | 2760.196931 |
| **...** | ... | ... | ... | ... | ... | ... |
| **1699** | Zimbabwe | Africa | 1987 | 62.351 | 9216418 | 706.157306 |
| **1700** | Zimbabwe | Africa | 1992 | 60.377 | 10704340 | 693.420786 |

877 rows × 6 columns

* We **only get those rows for which lifeExp > 60**
* If we **save the result back in df**, **all other rows will be deleted**
* **Only rows which satisfy the condition will remain**

Another way of accessing it is:

df[df.lifeExp > 60]

|  | **country** | **continent** | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- | --- | --- |
| **14** | Albania | Europe | 1962 | 64.820 | 1728137 | 2312.888958 |
| **15** | Albania | Europe | 1967 | 66.220 | 1984060 | 2760.196931 |
| **...** | ... | ... | ... | ... | ... | ... |
| **1699** | Zimbabwe | Africa | 1987 | 62.351 | 9216418 | 706.157306 |
| **1700** | Zimbabwe | Africa | 1992 | 60.377 | 10704340 | 693.420786 |

877 rows × 6 columns

**But this is not recommended. Why ?**

* It can create a confusion between implicit/explicit indexing used as discussed before
* loc is also much faster

**So, this can be another way of deleting rows based on certain conditions**

* But **be careful** while **saving the result** of filtering **in same variable vs. different variable**
* We might **still need original data to work with**

**Now this is how we can start querying our data based on the conditions we want**

**We can also pass in conditions for multiple columns at the same time**

* Works **just like slicing**

df.loc[df.lifeExp>60, ['country','lifeExp']]

# These will be the only 2 columns printed out

|  | **country** | **lifeExp** |
| --- | --- | --- |
| **14** | Albania | 64.820 |
| **15** | Albania | 66.220 |
| **16** | Albania | 67.690 |
| **17** | Albania | 68.930 |
| **18** | Albania | 70.420 |
| **...** | ... | ... |
| **1678** | Yemen, Rep. | 60.308 |
| **1679** | Yemen, Rep. | 62.698 |
| **1698** | Zimbabwe | 60.363 |
| **1699** | Zimbabwe | 62.351 |
| **1700** | Zimbabwe | 60.377 |

877 rows × 2 columns

# Printing only the first 5 rows of result now

df.loc[df.lifeExp>60, ['country','lifeExp']].head()

|  | **country** | **lifeExp** |
| --- | --- | --- |
| **14** | Albania | 64.82 |
| **15** | Albania | 66.22 |
| **16** | Albania | 67.69 |
| **17** | Albania | 68.93 |
| **18** | Albania | 70.42 |

* So far we saw only single condition for rows and columns

**What if we have multiple conditions to filter rows?**

**For Example, What if we want to filter data with lifeExp b/w 30 and 40?**

* In SQL: SELECT \* FROM df WHERE lifeExp<40 AND lifeExp>30
* We can **use AND operator b/w multiple conditions**

**But what is the AND operator in Pandas?**

* In **Python**, AND operator we saw is **and**
* But we cannot use **and** with pandas dataframe

**Why can’t we use and ?**

* Since a dataframe has multiple values
* and requires that both the values being compared are boolean
* In a df we would we getting a series of boolean values

df["lifeExp"] < 60 and df["lifeExp"] > 30

---------------------------------------------------------------------------

ValueError: The truth value of a Series is ambiguous. Use a.empty, a.bool(), a.item(), a.any() or a.all().

As you can see we get a value error in this case

**What should we do then ?**

* Use elementwise & operator
* **OR operator in Pandas is a single |**

df.loc[df.lifeExp>=30 & df.lifeExp<=40 ].head()

---------------------------------------------------------------------------

TypeError: ufunc 'bitwise\_and' not supported for the input types, and the inputs could not be safely coerced to any supported types according to the casting rule ''safe''

**But we ran into an error. Why?**

* Pandas require that **when multiple conditions are combined**
* We need to **put each separate condition within parenthesis ()**
* As soon as we put each condition in **separate ()**, it works!!

df.loc[ (df.lifeExp>=30) & (df.lifeExp<=40) ].head()

|  | **country** | **continent** | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- | --- | --- |
| **1** | Afghanistan | Asia | 1957 | 30.332 | 9240934 | 820.853030 |
| **2** | Afghanistan | Asia | 1962 | 31.997 | 10267083 | 853.100710 |
| **3** | Afghanistan | Asia | 1967 | 34.020 | 11537966 | 836.197138 |
| **4** | Afghanistan | Asia | 1972 | 36.088 | 13079460 | 739.981106 |
| **5** | Afghanistan | Asia | 1977 | 38.438 | 14880372 | 786.113360 |

**Question: Can you print out rows where country is either ‘Kenya’ or ‘Egypt’?**

df.loc[(df.country=='Kenya') | (df.country=='Egypt')]

|  | **country** | **continent** | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- | --- | --- |
| **456** | Egypt | Africa | 1952 | 41.893 | 22223309 | 1418.822445 |
| **457** | Egypt | Africa | 1957 | 44.444 | 25009741 | 1458.915272 |
| **458** | Egypt | Africa | 1962 | 46.992 | 28173309 | 1693.335853 |
| **459** | Egypt | Africa | 1967 | 49.293 | 31681188 | 1814.880728 |
| **460** | Egypt | Africa | 1972 | 51.137 | 34807417 | 2024.008147 |
| **461** | Egypt | Africa | 1977 | 53.319 | 38783863 | 2785.493582 |
| **462** | Egypt | Africa | 1982 | 56.006 | 45681811 | 3503.729636 |
| **463** | Egypt | Africa | 1987 | 59.797 | 52799062 | 3885.460710 |
| **464** | Egypt | Africa | 1992 | 63.674 | 59402198 | 3794.755195 |
| **465** | Egypt | Africa | 1997 | 67.217 | 66134291 | 4173.181797 |
| **466** | Egypt | Africa | 2002 | 69.806 | 73312559 | 4754.604414 |
| **467** | Egypt | Africa | 2007 | 71.338 | 80264543 | 5581.180998 |
| **816** | Kenya | Africa | 1952 | 42.270 | 6464046 | 853.540919 |
| **817** | Kenya | Africa | 1957 | 44.686 | 7454779 | 944.438315 |
| **818** | Kenya | Africa | 1962 | 47.949 | 8678557 | 896.966373 |
| **819** | Kenya | Africa | 1967 | 50.654 | 10191512 | 1056.736457 |
| **820** | Kenya | Africa | 1972 | 53.559 | 12044785 | 1222.359968 |
| **821** | Kenya | Africa | 1977 | 56.155 | 14500404 | 1267.613204 |
| **822** | Kenya | Africa | 1982 | 58.766 | 17661452 | 1348.225791 |
| **823** | Kenya | Africa | 1987 | 59.339 | 21198082 | 1361.936856 |
| **824** | Kenya | Africa | 1992 | 59.285 | 25020539 | 1341.921721 |
| **825** | Kenya | Africa | 1997 | 54.407 | 28263827 | 1360.485021 |
| **826** | Kenya | Africa | 2002 | 50.992 | 31386842 | 1287.514732 |
| **827** | Kenya | Africa | 2007 | 54.110 | 35610177 | 1463.249282 |

**Another one: Get all the countries which are alphabetically before ‘Nigeria’**

df.loc[df.country < 'Nigeria']

# String comparisons like this (>, <, ==) are also possible

|  | **country** | **continent** | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | Afghanistan | Asia | 1952 | 28.801 | 8425333 | 779.445314 |
| **1** | Afghanistan | Asia | 1957 | 30.332 | 9240934 | 820.853030 |
| **...** | ... | ... | ... | ... | ... | ... |
| **1126** | Niger | Africa | 2002 | 54.496 | 11140655 | 601.074501 |
| **1127** | Niger | Africa | 2007 | 56.867 | 12894865 | 619.676892 |

1128 rows × 6 columns

**Now lets say we want to know the average life expectancy of Asia**

df[df["continent"] == "Asia"]["lifeExp"].mean()

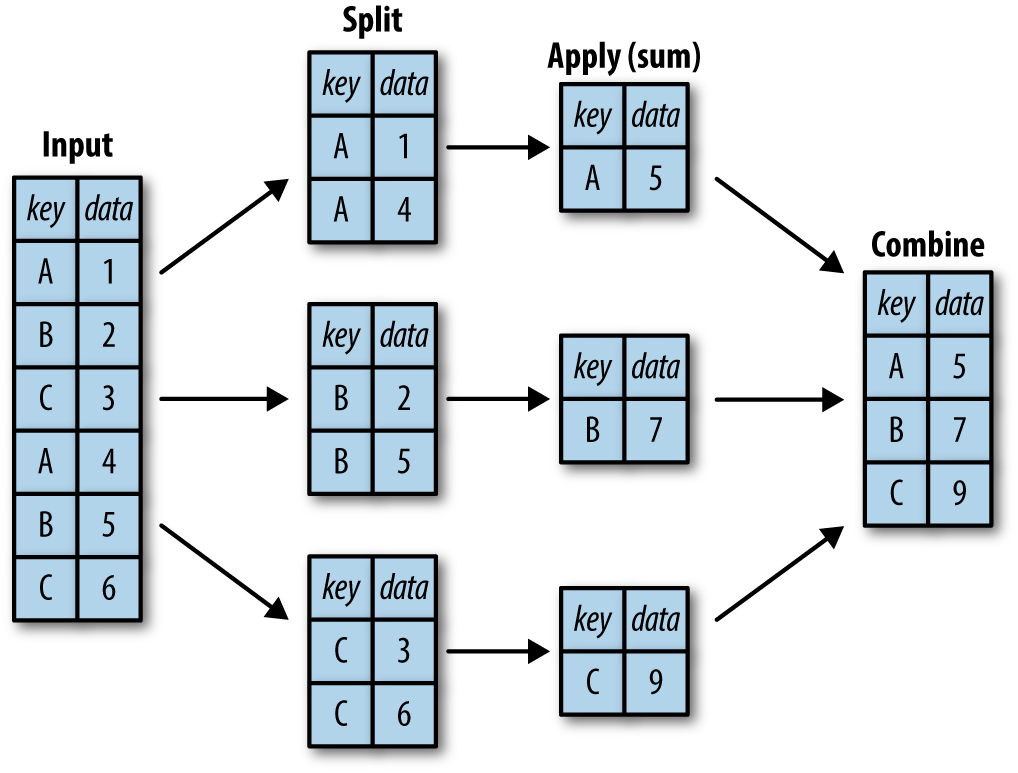
60.064903232323175

**What if we have to do this all possible continents?**

This can be be better solved by **Grouping**

**What is Grouping ?**

* Used to aggregate conditionally on some label or index
* In simpler words it could be understood through the terms: Split, apply, combine



* **Split**: Involves breaking up and grouping a DataFrame depending on the value of the specified key.
* **Apply**: Involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.
* **Combine**: Merges the results of these operations into an output array.

**Now we want to group countries by continent**

df.groupby('continent')

<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7efc228a8850>

* As you can see, it’s a **DataFrameGroupBy type object**
* **NOT a DataFrame** type object

**What is groupby('continent') doing?**

* It is **grouping all rows in which continent value is same**
* All the **rows having same continent will be grouped together**

**Now we want only 1 column (lifeExp) from the result of grouping and take their mean**

df.groupby('continent')[['lifeExp']].mean()

|  | **lifeExp** |
| --- | --- |
| **continent** |  |
| **Africa** | 48.865330 |
| **Americas** | 64.658737 |
| **Asia** | 60.064903 |
| **Europe** | 71.903686 |
| **Oceania** | 74.326208 |

Similalry we can also find the mean of lifeExp and gdpPerCap

df.groupby('continent')[['lifeExp', 'gdpPerCap']].mean()

|  | **lifeExp** | **gdpPerCap** |
| --- | --- | --- |
| **continent** |  |  |
| **Africa** | 48.865330 | 2193.754578 |
| **Americas** | 64.658737 | 7136.110356 |
| **Asia** | 60.064903 | 7902.150428 |
| **Europe** | 71.903686 | 14469.475533 |
| **Oceania** | 74.326208 | 18621.609223 |

**Now, What’s happening here?**

* First, all rows are grouped based on the values of 'continent'
* Then, only two columns ['lifeExp','gdpPerCap'] are selected for printing
* Finally, when we call mean() on it
* It **calculates** and returns **mean of values in those selected columns**,
* **grouped for each year**
* That is, **mean of values belonging to each year**

**Can we somehow see number of rows in each year group?**

* Just use **count()** instead of mean()
* The **operations** mean() and count() **don’t get applied on all values**
* **They get applied on each group** - because of df.groupby()

df.groupby('continent')[['lifeExp','gdpPerCap']].count()

|  | **lifeExp** | **gdpPerCap** |
| --- | --- | --- |
| **continent** |  |  |
| **Africa** | 624 | 624 |
| **Americas** | 300 | 300 |
| **Asia** | 396 | 396 |
| **Europe** | 360 | 360 |
| **Oceania** | 24 | 24 |

**Now what if we want to know the avg of lifeExp of different continents for each year ?**

* To do so we would have to group data acc to both continent and year

**How can we do that ?**

* Just **pass those column names packed in a list []**

df.groupby(['continent', 'year'])[['lifeExp','gdpPerCap']].mean()

|  |  | **lifeExp** | **gdpPerCap** |
| --- | --- | --- | --- |
| **continent** | **year** |  |  |
| **Africa** | **1952** | 39.135500 | 1252.572466 |
| **1957** | 41.266346 | 1385.236062 |
| **1962** | 43.319442 | 1598.078825 |
| **1967** | 45.334538 | 2050.363801 |
| **1972** | 47.450942 | 2339.615674 |
| **1977** | 49.580423 | 2585.938508 |
| **1982** | 51.592865 | 2481.592960 |
| **1987** | 53.344788 | 2282.668991 |
| **1992** | 53.629577 | 2281.810333 |
| **1997** | 53.598269 | 2378.759555 |
| **2002** | 53.325231 | 2599.385159 |
| **2007** | 54.806038 | 3089.032605 |
| **Americas** | **1952** | 53.279840 | 4079.062552 |
| **1957** | 55.960280 | 4616.043733 |
| **1962** | 58.398760 | 4901.541870 |
| **1967** | 60.410920 | 5668.253496 |
| **1972** | 62.394920 | 6491.334139 |
| **1977** | 64.391560 | 7352.007126 |
| **1982** | 66.228840 | 7506.737088 |
| **1987** | 68.090720 | 7793.400261 |
| **1992** | 69.568360 | 8044.934406 |
| **1997** | 71.150480 | 8889.300863 |
| **2002** | 72.422040 | 9287.677107 |
| **2007** | 73.608120 | 11003.031625 |
| **Asia** | **1952** | 46.314394 | 5195.484004 |
| **1957** | 49.318544 | 5787.732940 |
| **1962** | 51.563223 | 5729.369625 |
| **1967** | 54.663640 | 5971.173374 |
| **1972** | 57.319269 | 8187.468699 |
| **1977** | 59.610556 | 7791.314020 |
| **1982** | 62.617939 | 7434.135157 |
| **1987** | 64.851182 | 7608.226508 |
| **1992** | 66.537212 | 8639.690248 |
| **1997** | 68.020515 | 9834.093295 |
| **2002** | 69.233879 | 10174.090397 |
| **2007** | 70.728485 | 12473.026870 |
| **Europe** | **1952** | 64.408500 | 5661.057435 |
| **1957** | 66.703067 | 6963.012816 |
| **1962** | 68.539233 | 8365.486814 |
| **1967** | 69.737600 | 10143.823757 |
| **1972** | 70.775033 | 12479.575246 |
| **1977** | 71.937767 | 14283.979110 |
| **1982** | 72.806400 | 15617.896551 |
| **1987** | 73.642167 | 17214.310727 |
| **1992** | 74.440100 | 17061.568084 |
| **1997** | 75.505167 | 19076.781802 |
| **2002** | 76.700600 | 21711.732422 |
| **2007** | 77.648600 | 25054.481636 |
| **Oceania** | **1952** | 69.255000 | 10298.085650 |
| **1957** | 70.295000 | 11598.522455 |
| **1962** | 71.085000 | 12696.452430 |
| **1967** | 71.310000 | 14495.021790 |
| **1972** | 71.910000 | 16417.333380 |
| **1977** | 72.855000 | 17283.957605 |
| **1982** | 74.290000 | 18554.709840 |
| **1987** | 75.320000 | 20448.040160 |
| **1992** | 76.945000 | 20894.045885 |
| **1997** | 78.190000 | 24024.175170 |
| **2002** | 79.740000 | 26938.778040 |
| **2007** | 80.719500 | 29810.188275 |

**As we can see, grouping is happening on 2 levels**

* First, it’s **grouping based on 'continent'**
* Then, **within each 'continent'**, it is **grouping based on 'year'**

**What if we reverse the order of grouping?**

df.groupby(['year', 'continent'])[['lifeExp','gdpPerCap']].mean()

|  |  | **lifeExp** | **gdpPerCap** |
| --- | --- | --- | --- |
| **year** | **continent** |  |  |
| **1952** | **Africa** | 39.135500 | 1252.572466 |
| **Americas** | 53.279840 | 4079.062552 |
| **Asia** | 46.314394 | 5195.484004 |
| **Europe** | 64.408500 | 5661.057435 |
| **Oceania** | 69.255000 | 10298.085650 |
| **1957** | **Africa** | 41.266346 | 1385.236062 |
| **Americas** | 55.960280 | 4616.043733 |
| **Asia** | 49.318544 | 5787.732940 |
| **Europe** | 66.703067 | 6963.012816 |
| **Oceania** | 70.295000 | 11598.522455 |
| **1962** | **Africa** | 43.319442 | 1598.078825 |
| **Americas** | 58.398760 | 4901.541870 |
| **Asia** | 51.563223 | 5729.369625 |
| **Europe** | 68.539233 | 8365.486814 |
| **Oceania** | 71.085000 | 12696.452430 |
| **1967** | **Africa** | 45.334538 | 2050.363801 |
| **Americas** | 60.410920 | 5668.253496 |
| **Asia** | 54.663640 | 5971.173374 |
| **Europe** | 69.737600 | 10143.823757 |
| **Oceania** | 71.310000 | 14495.021790 |
| **1972** | **Africa** | 47.450942 | 2339.615674 |
| **Americas** | 62.394920 | 6491.334139 |
| **Asia** | 57.319269 | 8187.468699 |
| **Europe** | 70.775033 | 12479.575246 |
| **Oceania** | 71.910000 | 16417.333380 |
| **1977** | **Africa** | 49.580423 | 2585.938508 |
| **Americas** | 64.391560 | 7352.007126 |
| **Asia** | 59.610556 | 7791.314020 |
| **Europe** | 71.937767 | 14283.979110 |
| **Oceania** | 72.855000 | 17283.957605 |
| **1982** | **Africa** | 51.592865 | 2481.592960 |
| **Americas** | 66.228840 | 7506.737088 |
| **Asia** | 62.617939 | 7434.135157 |
| **Europe** | 72.806400 | 15617.896551 |
| **Oceania** | 74.290000 | 18554.709840 |
| **1987** | **Africa** | 53.344788 | 2282.668991 |
| **Americas** | 68.090720 | 7793.400261 |
| **Asia** | 64.851182 | 7608.226508 |
| **Europe** | 73.642167 | 17214.310727 |
| **Oceania** | 75.320000 | 20448.040160 |
| **1992** | **Africa** | 53.629577 | 2281.810333 |
| **Americas** | 69.568360 | 8044.934406 |
| **Asia** | 66.537212 | 8639.690248 |
| **Europe** | 74.440100 | 17061.568084 |
| **Oceania** | 76.945000 | 20894.045885 |
| **1997** | **Africa** | 53.598269 | 2378.759555 |
| **Americas** | 71.150480 | 8889.300863 |
| **Asia** | 68.020515 | 9834.093295 |
| **Europe** | 75.505167 | 19076.781802 |
| **Oceania** | 78.190000 | 24024.175170 |
| **2002** | **Africa** | 53.325231 | 2599.385159 |
| **Americas** | 72.422040 | 9287.677107 |
| **Asia** | 69.233879 | 10174.090397 |
| **Europe** | 76.700600 | 21711.732422 |
| **Oceania** | 79.740000 | 26938.778040 |
| **2007** | **Africa** | 54.806038 | 3089.032605 |
| **Americas** | 73.608120 | 11003.031625 |
| **Asia** | 70.728485 | 12473.026870 |
| **Europe** | 77.648600 | 25054.481636 |
| **Oceania** | 80.719500 | 29810.188275 |

**As we can see, the grouping gets reversed**

* Now, **First** it **groups based on 'year'**
* Then, **within each 'year'**, it does the **grouping based on each 'continent'**

Now this has created 2 levels in our dataframe

**What does this mean ?**

* Lets try and access the rows of df to understand this

year\_cont = df.groupby(['year', 'continent'])[['lifeExp','gdpPerCap']].mean()

year\_cont.loc[1952]

|  | **lifeExp** | **gdpPerCap** |
| --- | --- | --- |
| **continent** |  |  |
| **Africa** | 39.135500 | 1252.572466 |
| **Americas** | 53.279840 | 4079.062552 |
| **Asia** | 46.314394 | 5195.484004 |
| **Europe** | 64.408500 | 5661.057435 |
| **Oceania** | 69.255000 | 10298.085650 |

**Multi-Indexed Dataframes**

**We got multiple rows. But why ?**

* The dataframe created by groupby on 2 keys was a multi-index df

**Now what is Multi-index dataframe ?**

* It can have more than 2 dims like year\_cont that we created

**How can we access the rows in such a dataframe ?**

* By passing all the labels of all dims in a tuple
* Lets see it in code

year\_cont.loc[(1952, "Africa")]

lifeExp 39.135500

gdpPerCap 1252.572466

Name: (1952, Africa), dtype: float64

**How does iloc work with this though ?**

year\_cont.iloc[1]

lifeExp 53.279840

gdpPerCap 4079.062552

Name: (1952, Americas), dtype: float64

For iloc you just need to pass the python-style index of that row

**What if we want to get this grouping in 2D format?**

* We saw that the data type of this result of groupby() was DataFrameGroupBy

**So, what if we want to group our data, but bring the result back to a normal DataFrame?**

* We can **use reset\_index()**

dfgb = df.groupby(['continent','year'])[['lifeExp','gdpPerCap']].mean()

dfgb.reset\_index()

# And this is how our data looks like after applying reset\_index()

|  | **continent** | **year** | **lifeExp** | **gdpPerCap** |
| --- | --- | --- | --- | --- |
| **0** | Africa | 1952 | 39.135500 | 1252.572466 |
| **1** | Africa | 1957 | 41.266346 | 1385.236062 |
| **2** | Africa | 1962 | 43.319442 | 1598.078825 |
| **3** | Africa | 1967 | 45.334538 | 2050.363801 |
| **4** | Africa | 1972 | 47.450942 | 2339.615674 |
| **5** | Africa | 1977 | 49.580423 | 2585.938508 |
| **6** | Africa | 1982 | 51.592865 | 2481.592960 |
| **7** | Africa | 1987 | 53.344788 | 2282.668991 |
| **8** | Africa | 1992 | 53.629577 | 2281.810333 |
| **9** | Africa | 1997 | 53.598269 | 2378.759555 |
| **10** | Africa | 2002 | 53.325231 | 2599.385159 |
| **11** | Africa | 2007 | 54.806038 | 3089.032605 |
| **12** | Americas | 1952 | 53.279840 | 4079.062552 |
| **13** | Americas | 1957 | 55.960280 | 4616.043733 |
| **14** | Americas | 1962 | 58.398760 | 4901.541870 |
| **15** | Americas | 1967 | 60.410920 | 5668.253496 |
| **16** | Americas | 1972 | 62.394920 | 6491.334139 |
| **17** | Americas | 1977 | 64.391560 | 7352.007126 |
| **18** | Americas | 1982 | 66.228840 | 7506.737088 |
| **19** | Americas | 1987 | 68.090720 | 7793.400261 |
| **20** | Americas | 1992 | 69.568360 | 8044.934406 |
| **21** | Americas | 1997 | 71.150480 | 8889.300863 |
| **22** | Americas | 2002 | 72.422040 | 9287.677107 |
| **23** | Americas | 2007 | 73.608120 | 11003.031625 |
| **24** | Asia | 1952 | 46.314394 | 5195.484004 |
| **25** | Asia | 1957 | 49.318544 | 5787.732940 |
| **26** | Asia | 1962 | 51.563223 | 5729.369625 |
| **27** | Asia | 1967 | 54.663640 | 5971.173374 |
| **28** | Asia | 1972 | 57.319269 | 8187.468699 |
| **29** | Asia | 1977 | 59.610556 | 7791.314020 |
| **30** | Asia | 1982 | 62.617939 | 7434.135157 |
| **31** | Asia | 1987 | 64.851182 | 7608.226508 |
| **32** | Asia | 1992 | 66.537212 | 8639.690248 |
| **33** | Asia | 1997 | 68.020515 | 9834.093295 |
| **34** | Asia | 2002 | 69.233879 | 10174.090397 |
| **35** | Asia | 2007 | 70.728485 | 12473.026870 |
| **36** | Europe | 1952 | 64.408500 | 5661.057435 |
| **37** | Europe | 1957 | 66.703067 | 6963.012816 |
| **38** | Europe | 1962 | 68.539233 | 8365.486814 |
| **39** | Europe | 1967 | 69.737600 | 10143.823757 |
| **40** | Europe | 1972 | 70.775033 | 12479.575246 |
| **41** | Europe | 1977 | 71.937767 | 14283.979110 |
| **42** | Europe | 1982 | 72.806400 | 15617.896551 |
| **43** | Europe | 1987 | 73.642167 | 17214.310727 |
| **44** | Europe | 1992 | 74.440100 | 17061.568084 |
| **45** | Europe | 1997 | 75.505167 | 19076.781802 |
| **46** | Europe | 2002 | 76.700600 | 21711.732422 |
| **47** | Europe | 2007 | 77.648600 | 25054.481636 |
| **48** | Oceania | 1952 | 69.255000 | 10298.085650 |
| **49** | Oceania | 1957 | 70.295000 | 11598.522455 |
| **50** | Oceania | 1962 | 71.085000 | 12696.452430 |
| **51** | Oceania | 1967 | 71.310000 | 14495.021790 |
| **52** | Oceania | 1972 | 71.910000 | 16417.333380 |
| **53** | Oceania | 1977 | 72.855000 | 17283.957605 |
| **54** | Oceania | 1982 | 74.290000 | 18554.709840 |
| **55** | Oceania | 1987 | 75.320000 | 20448.040160 |
| **56** | Oceania | 1992 | 76.945000 | 20894.045885 |
| **57** | Oceania | 1997 | 78.190000 | 24024.175170 |
| **58** | Oceania | 2002 | 79.740000 | 26938.778040 |
| **59** | Oceania | 2007 | 80.719500 | 29810.188275 |

**With reset\_index():**

* We get our **columns in normal format**
* Each **row gets assigned a label number**
* The **values we got for lifeExp and gdpPerCap**, are now the **mean values for each continent and year**

**Now we want to find the countries that have been surveyed**

**How can we do that ?**

* For this, we need to find the unique vals in the country col
* Lets see how we can do that using pandas

df['country'].unique()

array(['Afghanistan', 'Albania', 'Algeria', 'Angola', 'Argentina',

'Australia', 'Austria', 'Bahrain', 'Bangladesh', 'Belgium',

'Benin', 'Bolivia', 'Bosnia and Herzegovina', 'Botswana', 'Brazil',

'Bulgaria', 'Burkina Faso', 'Burundi', 'Cambodia', 'Cameroon',

'Canada', 'Central African Republic', 'Chad', 'Chile', 'China',

'Colombia', 'Comoros', 'Congo, Dem. Rep.', 'Congo, Rep.',

'Costa Rica', "Cote d'Ivoire", 'Croatia', 'Cuba', 'Czech Republic',

'Denmark', 'Djibouti', 'Dominican Republic', 'Ecuador', 'Egypt',

'El Salvador', 'Equatorial Guinea', 'Eritrea', 'Ethiopia',

'Finland', 'France', 'Gabon', 'Gambia', 'Germany', 'Ghana',

'Greece', 'Guatemala', 'Guinea', 'Guinea-Bissau', 'Haiti',

'Honduras', 'Hong Kong, China', 'Hungary', 'Iceland', 'India',

'Indonesia', 'Iran', 'Iraq', 'Ireland', 'Israel', 'Italy',

'Jamaica', 'Japan', 'Jordan', 'Kenya', 'Korea, Dem. Rep.',

'Korea, Rep.', 'Kuwait', 'Lebanon', 'Lesotho', 'Liberia', 'Libya',

'Madagascar', 'Malawi', 'Malaysia', 'Mali', 'Mauritania',

'Mauritius', 'Mexico', 'Mongolia', 'Montenegro', 'Morocco',

'Mozambique', 'Myanmar', 'Namibia', 'Nepal', 'Netherlands',

'New Zealand', 'Nicaragua', 'Niger', 'Nigeria', 'Norway', 'Oman',

'Pakistan', 'Panama', 'Paraguay', 'Peru', 'Philippines', 'Poland',

'Portugal', 'Puerto Rico', 'Reunion', 'Romania', 'Rwanda',

'Sao Tome and Principe', 'Saudi Arabia', 'Senegal', 'Serbia',

'Sierra Leone', 'Singapore', 'Slovak Republic', 'Slovenia',

'Somalia', 'South Africa', 'Spain', 'Sri Lanka', 'Sudan',

'Swaziland', 'Sweden', 'Switzerland', 'Syria', 'Taiwan',

'Tanzania', 'Thailand', 'Togo', 'Trinidad and Tobago', 'Tunisia',

'Turkey', 'Uganda', 'United Kingdom', 'United States', 'Uruguay',

'Venezuela', 'Vietnam', 'West Bank and Gaza', 'Yemen, Rep.',

'Zambia', 'Zimbabwe'], dtype=object)

**Now lets find the total number of countries surveyed in each continent**

Step 1: Group the data acc to continent

Step 2: Apply nunique() method on that

df.groupby('continent')['country'].nunique()

continent

Africa 52

Americas 25

Asia 33

Europe 30

Oceania 2

Name: country, dtype: int64

**What do these values represent?**

* It tells us **No. of unique countries in every continent**
* First, it **grouped data based on continent**
* Then, it **counted unique values in country column for each continent**

We can also calculate multiple aggregates instead of just one

df.groupby(['continent','year'])[['lifeExp','gdpPerCap']].aggregate(['min', np.median, max])

|  |  | **lifeExp** | | | **gdpPerCap** | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **min** | **median** | **max** | **min** | **median** | **max** |
| **continent** | **year** |  |  |  |  |  |  |
| **Africa** | **1952** | 30.000 | 38.8330 | 52.724 | 298.846212 | 987.025569 | 4725.295531 |
| **1957** | 31.570 | 40.5925 | 58.089 | 335.997115 | 1024.022987 | 5487.104219 |
| **1962** | 32.767 | 42.6305 | 60.246 | 355.203227 | 1133.783678 | 6757.030816 |
| **1967** | 34.113 | 44.6985 | 61.557 | 412.977514 | 1210.376379 | 18772.751690 |
| **1972** | 35.400 | 47.0315 | 64.274 | 464.099504 | 1443.372508 | 21011.497210 |
| **1977** | 36.788 | 49.2725 | 67.064 | 502.319733 | 1399.638836 | 21951.211760 |
| **1982** | 38.445 | 50.7560 | 69.885 | 462.211415 | 1323.728306 | 17364.275380 |
| **1987** | 39.906 | 51.6395 | 71.913 | 389.876185 | 1219.585563 | 11864.408440 |
| **1992** | 23.599 | 52.4290 | 73.615 | 410.896824 | 1161.631357 | 13522.157520 |
| **1997** | 36.087 | 52.7590 | 74.772 | 312.188423 | 1179.883114 | 14722.841880 |
| **2002** | 39.193 | 51.2355 | 75.744 | 241.165876 | 1215.683217 | 12521.713920 |
| **2007** | 39.613 | 52.9265 | 76.442 | 277.551859 | 1452.267078 | 13206.484520 |
| **Americas** | **1952** | 37.579 | 54.7450 | 68.750 | 1397.717137 | 3048.302900 | 13990.482080 |
| **1957** | 40.696 | 56.0740 | 69.960 | 1544.402995 | 3780.546651 | 14847.127120 |
| **1962** | 43.428 | 58.2990 | 71.300 | 1662.137359 | 4086.114078 | 16173.145860 |
| **1967** | 45.032 | 60.5230 | 72.130 | 1452.057666 | 4643.393534 | 19530.365570 |
| **1972** | 46.714 | 63.4410 | 72.880 | 1654.456946 | 5305.445256 | 21806.035940 |
| **1977** | 49.923 | 66.3530 | 74.210 | 1874.298931 | 6281.290855 | 24072.632130 |
| **1982** | 51.461 | 67.4050 | 75.760 | 2011.159549 | 6434.501797 | 25009.559140 |
| **1987** | 53.636 | 69.4980 | 76.860 | 1823.015995 | 6360.943444 | 29884.350410 |
| **1992** | 55.089 | 69.8620 | 77.950 | 1456.309517 | 6618.743050 | 32003.932240 |
| **1997** | 56.671 | 72.1460 | 78.610 | 1341.726931 | 7113.692252 | 35767.433030 |
| **2002** | 58.137 | 72.0470 | 79.770 | 1270.364932 | 6994.774861 | 39097.099550 |
| **2007** | 60.916 | 72.8990 | 80.653 | 1201.637154 | 8948.102923 | 42951.653090 |
| **Asia** | **1952** | 28.801 | 44.8690 | 65.390 | 331.000000 | 1206.947913 | 108382.352900 |
| **1957** | 30.332 | 48.2840 | 67.840 | 350.000000 | 1547.944844 | 113523.132900 |
| **1962** | 31.997 | 49.3250 | 69.390 | 388.000000 | 1649.552153 | 95458.111760 |
| **1967** | 34.020 | 53.6550 | 71.430 | 349.000000 | 2029.228142 | 80894.883260 |
| **1972** | 36.088 | 56.9500 | 73.420 | 357.000000 | 2571.423014 | 109347.867000 |
| **1977** | 31.220 | 60.7650 | 75.380 | 371.000000 | 3195.484582 | 59265.477140 |
| **1982** | 39.854 | 63.7390 | 77.110 | 424.000000 | 4106.525293 | 33693.175250 |
| **1987** | 40.822 | 66.2950 | 78.670 | 385.000000 | 4106.492315 | 28118.429980 |
| **1992** | 41.674 | 68.6900 | 79.360 | 347.000000 | 3726.063507 | 34932.919590 |
| **1997** | 41.763 | 70.2650 | 80.690 | 415.000000 | 3645.379572 | 40300.619960 |
| **2002** | 42.129 | 71.0280 | 82.000 | 611.000000 | 4090.925331 | 36023.105400 |
| **2007** | 43.828 | 72.3960 | 82.603 | 944.000000 | 4471.061906 | 47306.989780 |
| **Europe** | **1952** | 43.585 | 65.9000 | 72.670 | 973.533195 | 5142.469716 | 14734.232750 |
| **1957** | 48.079 | 67.6500 | 73.470 | 1353.989176 | 6066.721495 | 17909.489730 |
| **1962** | 52.098 | 69.5250 | 73.680 | 1709.683679 | 7515.733737 | 20431.092700 |
| **1967** | 54.336 | 70.6100 | 74.160 | 2172.352423 | 9366.067033 | 22966.144320 |
| **1972** | 57.005 | 70.8850 | 74.720 | 2860.169750 | 12326.379990 | 27195.113040 |
| **1977** | 59.507 | 72.3350 | 76.110 | 3528.481305 | 14225.754515 | 26982.290520 |
| **1982** | 61.036 | 73.4900 | 76.990 | 3630.880722 | 15322.824720 | 28397.715120 |
| **1987** | 63.108 | 74.8150 | 77.410 | 3738.932735 | 16215.485895 | 31540.974800 |
| **1992** | 66.146 | 75.4510 | 78.770 | 2497.437901 | 17550.155945 | 33965.661150 |
| **1997** | 68.835 | 76.1160 | 79.390 | 3193.054604 | 19596.498550 | 41283.164330 |
| **2002** | 70.845 | 77.5365 | 80.620 | 4604.211737 | 23674.863230 | 44683.975250 |
| **2007** | 71.777 | 78.6085 | 81.757 | 5937.029526 | 28054.065790 | 49357.190170 |
| **Oceania** | **1952** | 69.120 | 69.2550 | 69.390 | 10039.595640 | 10298.085650 | 10556.575660 |
| **1957** | 70.260 | 70.2950 | 70.330 | 10949.649590 | 11598.522455 | 12247.395320 |
| **1962** | 70.930 | 71.0850 | 71.240 | 12217.226860 | 12696.452430 | 13175.678000 |
| **1967** | 71.100 | 71.3100 | 71.520 | 14463.918930 | 14495.021790 | 14526.124650 |
| **1972** | 71.890 | 71.9100 | 71.930 | 16046.037280 | 16417.333380 | 16788.629480 |
| **1977** | 72.220 | 72.8550 | 73.490 | 16233.717700 | 17283.957605 | 18334.197510 |
| **1982** | 73.840 | 74.2900 | 74.740 | 17632.410400 | 18554.709840 | 19477.009280 |
| **1987** | 74.320 | 75.3200 | 76.320 | 19007.191290 | 20448.040160 | 21888.889030 |
| **1992** | 76.330 | 76.9450 | 77.560 | 18363.324940 | 20894.045885 | 23424.766830 |
| **1997** | 77.550 | 78.1900 | 78.830 | 21050.413770 | 24024.175170 | 26997.936570 |
| **2002** | 79.110 | 79.7400 | 80.370 | 23189.801350 | 26938.778040 | 30687.754730 |
| **2007** | 80.204 | 80.7195 | 81.235 | 25185.009110 | 29810.188275 | 34435.367440 |

We can also calculate different aggregates for different cols like this:

df.groupby(['continent','year'])[['lifeExp','gdpPerCap']].aggregate({"lifeExp":'min',

"gdpPerCap": np.median})

|  |  | **lifeExp** | **gdpPerCap** |
| --- | --- | --- | --- |
| **continent** | **year** |  |  |
| **Africa** | **1952** | 30.000 | 987.025569 |
| **1957** | 31.570 | 1024.022987 |
| **1962** | 32.767 | 1133.783678 |
| **1967** | 34.113 | 1210.376379 |
| **1972** | 35.400 | 1443.372508 |
| **1977** | 36.788 | 1399.638836 |
| **1982** | 38.445 | 1323.728306 |
| **1987** | 39.906 | 1219.585563 |
| **1992** | 23.599 | 1161.631357 |
| **1997** | 36.087 | 1179.883114 |
| **2002** | 39.193 | 1215.683217 |
| **2007** | 39.613 | 1452.267078 |
| **Americas** | **1952** | 37.579 | 3048.302900 |
| **1957** | 40.696 | 3780.546651 |
| **1962** | 43.428 | 4086.114078 |
| **1967** | 45.032 | 4643.393534 |
| **1972** | 46.714 | 5305.445256 |
| **1977** | 49.923 | 6281.290855 |
| **1982** | 51.461 | 6434.501797 |
| **1987** | 53.636 | 6360.943444 |
| **1992** | 55.089 | 6618.743050 |
| **1997** | 56.671 | 7113.692252 |
| **2002** | 58.137 | 6994.774861 |
| **2007** | 60.916 | 8948.102923 |
| **Asia** | **1952** | 28.801 | 1206.947913 |
| **1957** | 30.332 | 1547.944844 |
| **1962** | 31.997 | 1649.552153 |
| **1967** | 34.020 | 2029.228142 |
| **1972** | 36.088 | 2571.423014 |
| **1977** | 31.220 | 3195.484582 |
| **1982** | 39.854 | 4106.525293 |
| **1987** | 40.822 | 4106.492315 |
| **1992** | 41.674 | 3726.063507 |
| **1997** | 41.763 | 3645.379572 |
| **2002** | 42.129 | 4090.925331 |
| **2007** | 43.828 | 4471.061906 |
| **Europe** | **1952** | 43.585 | 5142.469716 |
| **1957** | 48.079 | 6066.721495 |
| **1962** | 52.098 | 7515.733737 |
| **1967** | 54.336 | 9366.067033 |
| **1972** | 57.005 | 12326.379990 |
| **1977** | 59.507 | 14225.754515 |
| **1982** | 61.036 | 15322.824720 |
| **1987** | 63.108 | 16215.485895 |
| **1992** | 66.146 | 17550.155945 |
| **1997** | 68.835 | 19596.498550 |
| **2002** | 70.845 | 23674.863230 |
| **2007** | 71.777 | 28054.065790 |
| **Oceania** | **1952** | 69.120 | 10298.085650 |
| **1957** | 70.260 | 11598.522455 |
| **1962** | 70.930 | 12696.452430 |
| **1967** | 71.100 | 14495.021790 |
| **1972** | 71.890 | 16417.333380 |
| **1977** | 72.220 | 17283.957605 |
| **1982** | 73.840 | 18554.709840 |
| **1987** | 74.320 | 20448.040160 |
| **1992** | 76.330 | 20894.045885 |
| **1997** | 77.550 | 24024.175170 |
| **2002** | 79.110 | 26938.778040 |
| **2007** | 80.204 | 29810.188275 |

**Now lets say we want to find the countries being surveyed in a continent. How can we do that ?**

* By grouping the df using continent
* Returning the unique vals of country in the grouped df

df.groupby(["continent"])["country"].unique()

continent

Africa [Algeria, Angola, Benin, Botswana, Burkina Fas...

Americas [Argentina, Bolivia, Brazil, Canada, Chile, Co...

Asia [Afghanistan, Bahrain, Bangladesh, Cambodia, C...

Europe [Albania, Austria, Belgium, Bosnia and Herzego...

Oceania [Australia, New Zealand]

Name: country, dtype: object

**Filtering**

Apart from using aggregate functions, there are some other operations that can be performed with groupyby()

**Now we want to see data for continents having avg lifeExp < 50**

**How can we do this ?**

* We would first have to group the data acc to continent
* Then we need to calc avg lifeExp for each continent
* Finally, we need to separate data about continents have avg lifeExp < 50
* This process of dropping data based on the group properties is known as **Filtering**

df

|  | **country** | **continent** | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | Afghanistan | Asia | 1952 | 28.801 | 8425333 | 779.445314 |
| **1** | Afghanistan | Asia | 1957 | 30.332 | 9240934 | 820.853030 |
| **...** | ... | ... | ... | ... | ... | ... |
| **1702** | Zimbabwe | Africa | 2002 | 39.989 | 11926563 | 672.038623 |
| **1703** | Zimbabwe | Africa | 2007 | 43.487 | 12311143 | 469.709298 |

1704 rows × 6 columns

df.groupby('continent').mean()

|  | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- |
| **continent** |  |  |  |  |
| **Africa** | 1979.5 | 48.865330 | 9.916003e+06 | 2193.754578 |
| **Americas** | 1979.5 | 64.658737 | 2.450479e+07 | 7136.110356 |
| **Asia** | 1979.5 | 60.064903 | 7.703872e+07 | 7902.150428 |
| **Europe** | 1979.5 | 71.903686 | 1.716976e+07 | 14469.475533 |
| **Oceania** | 1979.5 | 74.326208 | 8.874672e+06 | 18621.609223 |

Now to filter values we need to create a filtering function

It will take a group as input and return some value

def filter\_func(x):

return x['lifeExp'].mean() < 50

Lets apply this function on our dataset now

df.groupby('continent').filter(filter\_func)

|  | **country** | **continent** | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- | --- | --- |
| **24** | Algeria | Africa | 1952 | 43.077 | 9279525 | 2449.008185 |
| **25** | Algeria | Africa | 1957 | 45.685 | 10270856 | 3013.976023 |
| **...** | ... | ... | ... | ... | ... | ... |
| **1702** | Zimbabwe | Africa | 2002 | 39.989 | 11926563 | 672.038623 |
| **1703** | Zimbabwe | Africa | 2007 | 43.487 | 12311143 | 469.709298 |

624 rows × 6 columns

**What is x in `filter\_func() though ?**

* Its a subframe of groups that are being created using groupby
* The function being implemented should return a logical output (i.e. True/False)
* Vals with output True will be kept and others will be dropped

**Transformation**

Sometimes you may want to “center” the data - bring the mean of the data to be zero

**We also want to subtract avg of a group’s (lets say country) lifeExp from lifeExp col of the data**

**How can we do that ?**

* Group data acc to country
* Calc its average lifeExp
* Subtract it from the data of that country
* This process of changing data is known as **Transformation**

def sub\_avg(x):

x["lifeExp"] -= x["lifeExp"].mean()

df.groupby(['country']).transform(sub\_avg)

---------------------------------------------------------------------------

KeyError: 'lifeExp'

**Now why did we get this error even though lifeExp col exists in our data ?**

* This is because the transform() function takes only 1 column of the df at a time and changes it
* Lets see what this means with a code.
* We will print the data type of x that transform receives

def inspect(x):

print(type(x))

raise

df.groupby(['country']).transform(inspect)

<class 'pandas.core.series.Series'>

---------------------------------------------------------------------------

RuntimeError: No active exception to reraise

**Ignore the runtime error**  
Note: We used raise to break the program

Look at the data type of x: pandas Series

Hence transform() can never work with 2 or more cols

**What should we do about our problem then ?**

* Just pass a series to transform

def sub\_avg(x):

x -= x.mean()

return x

df.groupby(['country'])[["lifeExp"]].transform(sub\_avg)

|  | **lifeExp** |
| --- | --- |
| **0** | -8.677833 |
| **1** | -7.146833 |
| **2** | -5.481833 |
| **3** | -3.458833 |
| **4** | -1.390833 |
| **...** | ... |
| **1699** | 9.687833 |
| **1700** | 7.713833 |
| **1701** | -5.854167 |
| **1702** | -12.674167 |
| **1703** | -9.176167 |

1704 rows × 1 columns

**But then what if we want to subtract 1 col’s mean from the other col ?**

* For eg we want to subtract gdpPerCap’s mean from lifeExp

This is where apply() comes into picture

**Apply**

**How can we do it ?**

* We again need to group data acc to country
* Subtracting mean of gdpPerCap from lifeExp
* To do so we would need to apply a custom function
* We can do so using the apply() method

def func(x):

x["lifeExp"] = x['lifeExp'] - x['gdpPerCap'].mean()

return x

df.groupby("country").apply(func)

|  | **country** | **continent** | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | Afghanistan | Asia | 1952 | -773.873598 | 8425333 | 779.445314 |
| **1** | Afghanistan | Asia | 1957 | -772.342598 | 9240934 | 820.853030 |
| **2** | Afghanistan | Asia | 1962 | -770.677598 | 10267083 | 853.100710 |
| **...** | ... | ... | ... | ... | ... | ... |
| **1703** | Zimbabwe | Africa | 2007 | -592.371042 | 12311143 | 469.709298 |

1704 rows × 6 columns

**Apply() takes a dataframe as the input**

Lets verfiy this once in the same way we did before

df.groupby("country").apply(inspect)

<class 'pandas.core.frame.DataFrame'>

---------------------------------------------------------------------------

RuntimeError : No active exception to reraise

**Ignore the error**

As you can see the data type is pandas dataframe

apply() can be applied on any dataframe along any particular axis

**What does this mean ?**

* The custom func will be apllied on each row if axis = 0 and on each col if axis = 1
* By default axis = 0
* Lets create a new dataframe to understand this

df[["lifeExp", "gdpPerCap"]].apply(np.sum, axis = 0)

lifeExp 1.013444e+05

gdpPerCap 1.229492e+07

dtype: float64

All rows in both cols were added

df[["lifeExp", "gdpPerCap"]].apply(np.sum, axis = 1)

0 808.246315

1 851.185030

2 885.097710

3 870.217138

4 776.069106

...

1699 768.508306

1700 753.797786

1701 839.258960

1702 712.027623

1703 513.196298

Length: 1704, dtype: float64

One row of lifeExp was added to same row of gdpPerCap

**Dealing with Missing Values using Pandas (just an Overview)**

**What are missing values?**

* A Missing Value means **there is nothing in that place**

**There can be 2 kinds of missing values:**

1. None
2. NaN (short for Not a Number)

**Whats the difference between the two ?**

* The diff mainly lies in their datatype

type(None)

NoneType

type(np.nan)

float

None type is for missing values in a column with non-number entries like strings

NaN occurs for columns with number entries

Pandas uses these values nearly interchangeably, converting between them where appropriate

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

0 1.0

1 NaN

2 2.0

3 NaN

dtype: float64

Now lets check some ways to deal with these missing values

**For this purpose, we will load a different dataset - disease.csv that we downloaded**

* Becasue gapminder.csv data did not have any missing values
* So, I am using this disease.csv dataset for demonstration

**So, Let’s read the file**

Link: <https://drive.google.com/file/d/1PP24ngaYsednG6y28N-y_9zq5hKEyTsu/view?usp=sharing>

!wget "https://drive.google.com/uc?export=download&id=1PP24ngaYsednG6y28N-y\_9zq5hKEyTsu" -O disease.csv

disease.csv 100%[===================>] 4.52K --.-KB/s in 0s

disease = pd.read\_csv('disease.csv')

**Notice the dataset**

* Some of the values are missing - NaN

**Why do we need to handle these missing values?**

* Because most ML and DS algorithms break when they encounter missing data
* Missing data depreciate the performance of our models

**Now, How can we deal with missing values? What ideas do you have?**

* It is on **case-to-case basis**
* We have to **pick a method based on the dataset and SITUATION**
* We have to **check what will work and what not**

**Ask yourself: What makes sense and what not?**

* **DON’T just remove OR replace with default 0 OR replace with mean OR anything else blindly**
* The way we choose to deal with missing values can be **easily misleading**

**So, we need to be very careful about how we choose to deal with missing data**

* Use something **simple**
* But it **should make sense in the given situation**

**There are many ways to deal with missing values**

**First, Let’s import NaN from Numpy**

* Missing value in data can appear in 3 ways
  + NaN
  + NAN
  + nan

from numpy import NaN, NAN, nan

# One interesting thing about Nan is this:

nan == nan

False

**Why did it come out be False?**

**Can you compare two infinite or non-existent values?**

* NO
* So, **be careful** while searching for missing values **using ==**

**So, How can we find if a value is missing or not?**

* We can use **is**
* We can use **Pandas in-built function isnull()**

nan is nan

True

pd.isnull(nan)

True

**Now coming back to our data**

**Let’s do some exploration before jumping into dealing with missing values**

* We can check **data.info()** to get an **idea of distribution of missing values**
* It gives **no. of non-null (Available) values** in each column

disease.info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 122 entries, 0 to 121

Data columns (total 14 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Date 122 non-null object

1 Day 122 non-null int64

2 Cases\_Guinea 93 non-null float64

3 Cases\_Liberia 83 non-null float64

4 Cases\_Nigeria 38 non-null float64

5 Cases\_Senegal 25 non-null float64

6 Cases\_UnitedKingdom 18 non-null float64

7 Cases\_Mali 12 non-null float64

8 Deaths\_Guinea 92 non-null float64

9 Deaths\_Liberia 81 non-null float64

10 Deaths\_Nigeria 38 non-null float64

11 Deaths\_Senegal 22 non-null float64

12 Deaths\_UnitedKingdom 18 non-null float64

13 Deaths\_Mali 12 non-null float64

dtypes: float64(12), int64(1), object(1)

memory usage: 13.5+ KB

We can also find the total number of NULL values in each column

disease.isna().sum()

Date 0

Day 0

Cases\_Guinea 29

Cases\_Liberia 39

Cases\_Nigeria 84

Cases\_Senegal 97

Cases\_UnitedKingdom 104

Cases\_Mali 110

Deaths\_Guinea 30

Deaths\_Liberia 41

Deaths\_Nigeria 84

Deaths\_Senegal 100

Deaths\_UnitedKingdom 104

Deaths\_Mali 110

dtype: int64

# Let's print head to see what's in the dataset again

disease.head()

|  | **Date** | **Day** | **Cases\_Guinea** | **Cases\_Liberia** | **Cases\_Nigeria** | **Cases\_Senegal** | **Cases\_UnitedKingdom** | **Cases\_Mali** | **Deaths\_Guinea** | **Deaths\_Liberia** | **Deaths\_Nigeria** | **Deaths\_Senegal** | **Deaths\_UnitedKingdom** | **Deaths\_Mali** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 7/22/2014 | 122 | 2770.0 | NaN | NaN | NaN | NaN | NaN | 1786.0 | NaN | NaN | NaN | NaN | NaN |
| **1** | 7/21/2014 | 121 | 2771.0 | NaN | NaN | NaN | NaN | NaN | 1781.0 | NaN | NaN | NaN | NaN | NaN |
| **2** | 7/20/2014 | 120 | 2765.0 | 8165.0 | NaN | NaN | NaN | NaN | 1767.0 | 3496.0 | NaN | NaN | NaN | NaN |
| **3** | 7/19/2014 | 119 | NaN | 8156.0 | NaN | NaN | NaN | NaN | NaN | 3496.0 | NaN | NaN | NaN | NaN |
| **4** | 7/18/2014 | 118 | 2730.0 | 8115.0 | NaN | NaN | NaN | NaN | 1739.0 | 3471.0 | NaN | NaN | NaN | NaN |

* We can get **frequency of each value** in a column
* **No. of occurrences of each value** in a column
* Using value\_counts()

disease['Cases\_Guinea'].value\_counts()

86.0 3

495.0 2

112.0 2

390.0 2

408.0 1

..

1199.0 1

1298.0 1

1350.0 1

1472.0 1

49.0 1

Name: Cases\_Guinea, Length: 88, dtype: int64

**It tells that**

* Value 86.0 occurs 3 times
* Value 112.0 occurs 2 times
* … and so on

**But its not telling the count of missing values**

* Because by **default**, the **parameter dropna is set to True**
* dropna=True means it is **NOT going to count missing values**
* So, **we have to set dropna=False** to get the **count of missing values in a column**

disease['Cases\_Guinea'].value\_counts(dropna=False)

NaN 29

86.0 3

495.0 2

112.0 2

390.0 2

..

1199.0 1

1298.0 1

1350.0 1

1472.0 1

49.0 1

Name: Cases\_Guinea, Length: 89, dtype: int64

* We see, the highest frequency is of Missing Values

**Now, How can we see the unique values?**

* **unique()** gives **unique values** in a column
* **nunique()** gives **number of unique values** in a column

u = disease['Cases\_Guinea'].unique()

u

array([2770., 2771., 2765., nan, 2730., 2706., 2695., 2630., 2597.,

2571., 2416., 2292., 2164., 2134., 2047., 1971., 1919., 1878.,

1770., 1731., 1667., 1906., 1553., 1540., 1519., 1472., 1350.,

1298., 1199., 1157., 1074., 1022., 1008., 942., 936., 899.,

861., 812., 771., 648., 607., 579., 543., 519., 510.,

506., 495., 485., 472., 460., 427., 415., 410., 411.,

406., 409., 408., 412., 413., 390., 398., 351., 344.,

328., 291., 281., 258., 248., 233., 236., 235., 231.,

226., 224., 218., 208., 203., 197., 168., 159., 158.,

151., 143., 127., 122., 112., 103., 86., 49.])

len(u)

89

disease['Cases\_Guinea'].nunique()

88

**Now, Why are unique() and nunique() giving different number of unique values?**

* unique() is counting NaN as well
* nunique() does not count NaN by default
* So, we have to set dropna=False for nunique()

disease['Cases\_Guinea'].nunique(dropna=False)

89

We can also check whether a particular value in df is null or not

disease.isnull()

|  | **Date** | **Day** | **Cases\_Guinea** | **Cases\_Liberia** | **Cases\_Nigeria** | **Cases\_Senegal** | **Cases\_UnitedKingdom** | **Cases\_Mali** | **Deaths\_Guinea** | **Deaths\_Liberia** | **Deaths\_Nigeria** | **Deaths\_Senegal** | **Deaths\_UnitedKingdom** | **Deaths\_Mali** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | False | False | False | True | True | True | True | True | False | True | True | True | True | True |
| **1** | False | False | False | True | True | True | True | True | False | True | True | True | True | True |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **121** | False | False | False | True | True | True | True | True | False | True | True | True | True | True |

122 rows × 14 columns

**Removing/Dropping the missing values**

**What if you have just 1 or very few rows which has missing data, compared to whole data?**

* **Example**: only 10 rows out of 1 million rows having missing values
* We can simply remove those rows or columns using **drop()** - already seen

**Replacing missing values with mean**

m = disease['Cases\_Guinea'].mean()

m

911.0215053763441

* **value= can be anything**
  + mean
  + median
  + mode
  + 0
  + min, max
  + …

**Backward and Forward fill approach**

* Let’s read the dataset file again

disease = pd.read\_csv('disease.csv')

* We can **use previous and/or next value** to fill the missing value

1. Take **average of previous and next value**
2. **Copy previous value** to missing value
3. **Copy next value** to missing value

* We use **fillna() method**

**Foreward Fill**

* Fills missing value **from previous row’s value**
* **Previous row is filling forward**

disease.fillna(method='ffill')

|  | **Date** | **Day** | **Cases\_Guinea** | **Cases\_Liberia** | **Cases\_Nigeria** | **Cases\_Senegal** | **Cases\_UnitedKingdom** | **Cases\_Mali** | **Deaths\_Guinea** | **Deaths\_Liberia** | **Deaths\_Nigeria** | **Deaths\_Senegal** | **Deaths\_UnitedKingdom** | **Deaths\_Mali** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 7/22/2014 | 122 | 2770.0 | NaN | NaN | NaN | NaN | NaN | 1786.0 | NaN | NaN | NaN | NaN | NaN |
| **1** | 7/21/2014 | 121 | 2771.0 | NaN | NaN | NaN | NaN | NaN | 1781.0 | NaN | NaN | NaN | NaN | NaN |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **121** | 3/23/2014 | 1 | 49.0 | 8.0 | 0.0 | 1.0 | 1.0 | 1.0 | 29.0 | 6.0 | 0.0 | 0.0 | 0.0 | 1.0 |

122 row

**Notice first 2 rows of Cases\_Liberia column**

* First 2 rows **still have NaN**
* **Forward Fill is not able to fill those values**
* Because there is **no previous value to help in filling those missing values**

**Backward Fill**

* Fills missing value **from next row’s value**
* **Next row is filling backwards**

disease.fillna(method='bfill')

|  | **Date** | **Day** | **Cases\_Guinea** | **Cases\_Liberia** | **Cases\_Nigeria** | **Cases\_Senegal** | **Cases\_UnitedKingdom** | **Cases\_Mali** | **Deaths\_Guinea** | **Deaths\_Liberia** | **Deaths\_Nigeria** | **Deaths\_Senegal** | **Deaths\_UnitedKingdom** | **Deaths\_Mali** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 7/22/2014 | 122 | 2770.0 | 8165.0 | 20.0 | 1.0 | 4.0 | 7.0 | 1786.0 | 3496.0 | 8.0 | 0.0 | 1.0 | 6.0 |
| **1** | 7/21/2014 | 121 | 2771.0 | 8165.0 | 20.0 | 1.0 | 4.0 | 7.0 | 1781.0 | 3496.0 | 8.0 | 0.0 | 1.0 | 6.0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **120** | 3/24/2014 | 2 | 86.0 | NaN | NaN | NaN | NaN | NaN | 59.0 | NaN | NaN | NaN | NaN | NaN |
| **121** | 3/23/2014 | 1 | 49.0 | NaN | NaN | NaN | NaN | NaN | 29.0 | NaN | NaN | NaN | NaN | NaN |

122 rows × 14 columns

**Notice in this case last few rows of Cases\_Liberia column**

* Last few rows **still have NaN**
* **Backward Fill is not able to fill those values**
* Because there is **no next value to help in filling those missing values**

**So, we can do forward and backward fill one after the other**

disease.fillna(method='ffill')

disease.fillna(method='bfill')

|  | **Date** | **Day** | **Cases\_Guinea** | **Cases\_Liberia** | **Cases\_Nigeria** | **Cases\_Senegal** | **Cases\_UnitedKingdom** | **Cases\_Mali** | **Deaths\_Guinea** | **Deaths\_Liberia** | **Deaths\_Nigeria** | **Deaths\_Senegal** | **Deaths\_UnitedKingdom** | **Deaths\_Mali** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 7/22/2014 | 122 | 2770.0 | 8165.0 | 20.0 | 1.0 | 4.0 | 7.0 | 1786.0 | 3496.0 | 8.0 | 0.0 | 1.0 | 6.0 |
| **1** | 7/21/2014 | 121 | 2771.0 | 8165.0 | 20.0 | 1.0 | 4.0 | 7.0 | 1781.0 | 3496.0 | 8.0 | 0.0 | 1.0 | 6.0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **121** | 3/23/2014 | 1 | 49.0 | NaN | NaN | NaN | NaN | NaN | 29.0 | NaN | NaN | NaN | NaN | NaN |

122 rows × 14 columns

**Let’s check head of our data**

disease.head()

|  | **Date** | **Day** | **Cases\_Guinea** | **Cases\_Liberia** | **Cases\_Nigeria** | **Cases\_Senegal** | **Cases\_UnitedKingdom** | **Cases\_Mali** | **Deaths\_Guinea** | **Deaths\_Liberia** | **Deaths\_Nigeria** | **Deaths\_Senegal** | **Deaths\_UnitedKingdom** | **Deaths\_Mali** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 7/22/2014 | 122 | 2770.0 | NaN | NaN | NaN | NaN | NaN | 1786.0 | NaN | NaN | NaN | NaN | NaN |
| **1** | 7/21/2014 | 121 | 2771.0 | NaN | NaN | NaN | NaN | NaN | 1781.0 | NaN | NaN | NaN | NaN | NaN |
| **2** | 7/20/2014 | 120 | 2765.0 | 8165.0 | NaN | NaN | NaN | NaN | 1767.0 | 3496.0 | NaN | NaN | NaN | NaN |
| **3** | 7/19/2014 | 119 | NaN | 8156.0 | NaN | NaN | NaN | NaN | NaN | 3496.0 | NaN | NaN | NaN | NaN |
| **4** | 7/18/2014 | 118 | 2730.0 | 8115.0 | NaN | NaN | NaN | NaN | 1739.0 | 3471.0 | NaN | NaN | NaN | NaN |

* **Actual Data is not changed**

**What could be the problem here?**

* You should be able to figure out now
* We need to **set inplace=True**

**This is all about Pandas for today**

* We’ll cover a few more important concepts in the next lecture

Last changed by

**Scaler Topics**Scaler Topics

 159

Published on [**HackMD**](https://hackmd.io/)

**Practice of Pandas Grouping**

Practice Question on Grouping

We already know what grouping is

Lets solve a question on it for our practice  
You are given the DataFrame below with a column of ‘grps’, and a column of corresponding integer values, ‘vals’.

Use the following code to generate it

df = pd.DataFrame({"vals": np.random.RandomState(31).randint(-30, 30, size=15),

"grps": np.random.RandomState(31).choice(["A", "B"], 15)})

Create a new column ‘patched\_values’

This column should we same as vals but replace the neg values with the mean of pos values in that group

The resultant df should look like this

vals grps patched\_vals

0 -12 A 13.6

1 -7 B 28.0

2 -14 A 13.6

3 4 A 4.0

4 -7 A 13.6

5 28 B 28.0

6 -2 A 13.6

7 -1 A 13.6

8 8 A 8.0

9 -2 B 28.0

10 28 A 28.0

11 12 A 12.0

12 16 A 16.0

13 -24 A 13.6

14 -12 A 13.6

Group A has 5 positive values with mean 13.6

**Solution:**

import pandas as pd

import numpy as np

quiz = pd.DataFrame({"vals": np.random.RandomState(31).randint(-30, 30, size=15),

"grps": np.random.RandomState(31).choice(["A", "B"], 15)})

def replace(group):

mask1 = group < 0

mask2 = group > 0

group[mask1] = group[mask2].mean()

return group

quiz.groupby(['grps'])['vals'].transform(replace)

0 13.6

1 28.0

2 13.6

3 4.0

4 13.6

5 28.0

6 13.6

7 13.6

8 8.0

9 28.0

10 28.0

11 12.0

12 16.0

13 13.6

14 13.6

Name: vals, dtype: float64

Last changed by

**Scaler Topics**Scaler Topics

 249

Published on [**HackMD**](https://hackmd.io/)