# BUSAN 300 - Python

# Set-Up

```
import pandas as pd
data = {
        "Name": ["Alice", "Bob", "Charlie"],
        "Age": [25, 30, 25],
        "City": ["New York", "Los Angeles", "Chicago"]
}
df = pd.DataFrame(data)          # Take dictionary and instantiate into a Data Frame
print(df)
```

## Mount Google Drive

```
from google.colab import drive
drive.mount('/content/drive')    # Mount Google Drive for Google Colab
```

## Read in dataset

```
df = pd.read_csv("content/drive/MyDrive/data.csv" , dtype='unicode')     # Read in dataset
```

# .loc[] and .iloc[]

**.loc[] - Locates by name (Label-based indexing)**
**.iloc[] - Locates by numerical index**

```
df.iloc(['Nami', 'Gender'])          # F
df.iloc(['Usopp'])                   # 'Usopp', 18, 'M', 3
df.iloc([:, 'Gender'])               # 'M', 'F', 'M', 'M'
df.iloc(['Chopper', 'Rating'])       # 4

df.loc([0, 0])                       # 'Luffy'
df.loc([0])                          # 'Luffy', 17, 'M', 5
df.loc([:, 1])                       # 17, 18, 18, 15
df.loc([0:1])                        # ['Luffy', 17, 'M', 5], ['Nami', 18, 'F', 4]
```

# Simple Queries

## Select a column

```
complaints = pd.read_csv('/content/drive/data.csv', dtype='unicode')
complaints['Complaint Type']
```

## Sort

```
complaints = complaints.sort_values(by="Created Date", ascending=True)
```

## Pulling & Counting Unique Values

```
complaints['Complaint Type'].unique()
complaints['Complaint Type'].nunique()
```

## Pulling the most frequent

```
complaints['Complaint Type'].value_counts()
complaints['Complaint Type', 'City'].value_counts() # Count unique complaint type and which city
```

| Complaint Type | City | |
|---|---|---|
| HEATING | BROOKLYN | 226 |
| | BRONX | 208 |
| GENERAL CONSTRUCTION | BRONX | 125 |
| HEATING | NEW YORK | 118 |
| PLUMBING | BROOKLYN | 107 |
| ... | ... | ... |

*Count how many complaints of each type happened in each city.*

We see that the most frequent complaint type is HEATING, and this complaint is most prominent in the City of Brooklyn.

## Extracting entries with conditions

```
# From the complaints dataframe, subset this to include only the column Complaint Types where
those entries have the word HEAT or heat in them, and ignore NA's.
complaints[complaints['Complaint Type'].str.contains('heat', case=False, na=False)]

# From the complaints dataframe, subset this to include column Complaint Types where those
entries have the word heat/HEAT or rodent/RODENT in them, and ignore NA's
complaints[complaints['Complaint Type'].str.contains('heat | rodent', case=False, na=False)]

# Assign the above to a variable called 'heat_complaints'
heat_complaints = complaints[
                    complaints['Complaint Type'].str.contains('heat', case=False, na=False)]

# From all heating complaints, extract entries where the complaint type isn't 'HEATING'
heat_complaints[heat_complaints['Complaint Type'] != 'HEATING']

# Extract noise complaints and count how many there are
noise_complaints = complaints[complaints['Complaint Type'] == "Noise - Street/Sidewalk"]
print(len(complaints))
print(len(noise_complaints))

# Find and count noise complaints in the Bronx
noise_complaints_bronx = noise_complaints[noise_complaints['City'] == "BRONX"]
print(len(noise_complaints_bronx))

# Specify the columns we want to see as a temporary view
noise_complaints_bronx[['Complaint Type', 'City', 'Created Date', 'Descriptor']]

# Finding the City with the MOST noise complaints
is_noise = complaints['Complaint Type'] == 'Noise - Street/Sidewalk'
noise_complaints = complaints[is_noise]
noise_complaints['City'].value_counts()

# Plot the noise complaints by City
noise_complaints['City'].value_counts().plot(kind='bar')
```

# Personally Identifiable Information

```
# Using nested JSON data, normalising, and converting into a Data Frame
df_nested = pd.json_normalize(data)
```

| | customer.name | customer.email | customer.phone | review.restaurant | review.rating | review.t |
|---|---|---|---|---|---|---|
| 0 | Emma Carter | emma.c@example.com | 555-1001 | The Toasted Fork | 5 | Service was quick, and the grilled cheese wa |
| 1 | Liam Brooks | liam.b@example.com | 555-1002 | Bun & Barrel | 4 | Great burgers, and the milkshakes brought ba |
| 2 | Sophie Kim | sophie.k@example.com | 555-1003 | Sushi Lane | 3 | Fresh ingredients, but the service was a bit |
| 3 | Noah Reed | noah.r@example.com | 555-1004 | Taco Station | 4 | Tasty tacos with a fun street-food vibe. Wou |

```
# Rename the column names for clarity using .columns
df_nested.columns = ['Customer_Name', 'Email', 'Phone', 'Restaurant', 'Rating', 'Review']
```

| | Customer_Name | Email | Phone | Restaurant | Rating | Review |
|---|---|---|---|---|---|---|
| 0 | Emma Carter | emma.c@example.com | 555-1001 | The Toasted Fork | 5 | Service was quick, and the grilled cheese was ... |
| 1 | Liam Brooks | liam.b@example.com | 555-1002 | Bun & Barrel | 4 | Great burgers, and the milkshakes brought back... |
| 2 | Sophie Kim | sophie.k@example.com | 555-1003 | Sushi Lane | 3 | Fresh ingredients, but the service was a bit s... |
| 3 | Noah Reed | noah.r@example.com | 555-1004 | Taco Station | 4 | Tasty tacos with a fun street-food vibe. Would... |

```
# Anonymise the data
# STEP 1: Create a new column 'Reviewer_ID' and create an ID per cell
df_nested['Reviewer_ID'] = ['Reviewer_' + str(i + 1) for i in df_nested.index]
# STEP 2: Remove the identifying rows
df_anonymised = df_nested.drop(columns=['Customer_Name', 'Email', 'Phone'])
```

| | Restaurant | Rating | Review | Reviewer_ID |
|---|---|---|---|---|
| 0 | The Toasted Fork | 5 | Service was quick, and the grilled cheese was ... | Reviewer_1 |
| 1 | Bun & Barrel | 4 | Great burgers, and the milkshakes brought back... | Reviewer_2 |
| 2 | Sushi Lane | 3 | Fresh ingredients, but the service was a bit s... | Reviewer_3 |
| 3 | Taco Station | 4 | Tasty tacos with a fun street-food vibe. Would... | Reviewer_4 |

It is relatively easy to identify individuals even after analysing it, for example, copying and pasting their review text into google. Hence, we can attempt to obfuscate the text.

```
# Remove commas from "Review" using .str.replace()
df_anonymised['Review'] = df_anonymised['Review'].str.replace(',', '')
```

# Date Modification

```
# Check types
print(complaints['Created Date'].dtype)
```

```
# Convert 'Created Date' into a datetime object
```

```
complaints['Created Date'] = pd.to_datetime(complaints['Created Date'])
print(complaints['Created Date'].dtype)

# Extract dates Year, Month and Day using dt.year, dt.month, dt.day
complaints['Year'] = complaints['Created Date'].dt.year
complaints['Month'] = complaints['Created Date'].dt.month
complaints['Day'] = complaints['Created Date'].dt.day

# Extract date in a new format
complaints['Date'] = complaints['Created Date'].dt.strftime('%d-%m-%Y')

'%d-%m-%Y'     # 29-05-2025
'%Y-%m-%d'     # 2025-05-29
'%B %d, %Y'    # May 29, 2025
'%m/%d/%y'     # 05/29/25
'%A'           # Day of the week, e.g. Thursday
'%I:%M %p'     # 12-hour time, e.g. 03:15 PM
```

```
# %d   - Day of the month (01 to 31)
# %m   - Month number (01 to 12)
# %b   - Abbreviated month name (Jan, Feb, ...)
# %B   - Full month name (January, February, ...)
# %y   - 2-digit year (25)
# %Y   - 4-digit year (2025)

# %H   - Hour (00 to 23, 24-hour clock)
# %I   - Hour (01 to 12, 12-hour clock)
# %p   - AM or PM
# %M   - Minute (00 to 59)
# %S   - Second (00 to 59)

# %a   - Abbreviated weekday name (Mon, Tue, ...)
# %A   - Full weekday name (Monday, Tuesday, ...)
# %w   - Weekday as a number (0 = Sunday, 6 = Saturday)
# %j   - Day of the year (001 to 366)
# %U   - Week number (Sunday as first day of week, 00 to 53)
# %W   - Week number (Monday as first day of week, 00 to 53)
```

```
# Interstly, if we extract all the Date counts, we only get two unique dates
complaints['Date'].value_counts()
```

| Date | |
|---|---|
| 30-10-2013 | 4933 |
| 31-10-2013 | 67 |

```
# We can filter complaints from a specific year
```

```
complaints['Created Date'].pd.to_datetime(complaints['Created Date'])
complaints['Year'] = complaints['Created Date'].dt.year
complaints_2010 = complaints[complaints['Year'] == '2010']
print(len(complaints_2010))

# Calculate the difference between 'Created Date' and 'Closed Date'
complaints['Created Date'].pd.to_datetime(complaints['Created Date'])
complaints['Closed Date'].pd.to_datetime(complaints['Closed Date'])
complaints['Time Difference'] = complaints['Created Date'] - complaints['Closed Date']

# Display complaints with a resolution time greater than 1 day
long_resolutions = complaints[complaints['Time Difference'] > pd.Timedelta(days = 1)]

# Alternatively, finding complaints with resolution times greater than 1 minute, hour, week
long_resolutions = complaints[complaints['Time Difference'] > pd.Timedelta(minutes = 1)]
long_resolutions = complaints[complaints['Time Difference'] > pd.Timedelta(hours = 1)]
long_resolutions = complaints[complaints['Time Difference'] > pd.Timedelta(weeks = 1)]

# Group by Date and find the count of each unique date
complaints.groupby('Date').count()
```

# Regular Expressions (RegEx)

| | | |
|---|---|---|
| abc | Matches "abc" in "abc123" | |
| a.c | Matches "abc" in "abc123" | |
| ^abc | Matches "abc" at the beginning of "abc123" | |
| 123$ | Matches "123" at the end of "abc123" | |
| ab*c | Matches "ac", "abc", "abbc" | (You can have 'b' anywhere 0, 1, 2, etc times) |
| ab+c | Matches "abc" in "abbc" | (You can have 'b' anywhere 1, 2, etc times) |
| ab?c | Matches "ac", "abc" | (You can have 'b' anywhere 0 or 1 times) |
| [abc] | Matches "a", "b", or "c" | |
| ` | `abc | (OR operator) |
| (abc)+ | Matches "abc", "abcabc" | (Groups patterns together) |

## Removing <HTML> Tags

```
# STEP 1: Finding all '<' in a string using re.findall()
import re
text = "<p>This product is <b>amazing</b>!</p>"
pattern = r"<"                          # When specifying a RegEx pattern, it has to begin with r
matches = re.findall(pattern, text)          # ['<', '<', '<', '<']
```

*Square brackets, [ ], define a character class, that is, anything inside the [ ] define our match criteria. Although, if we use a ^, it NEGATES the characters class specified in [ ]. Hence,* r"[^<]" means our character class is '<', but we negate using ^, so anything that ISN'T '<' is a match.

```
# STEP 2: Finding all '<' in a string using re.findall()
text = "<p>This product is <b>amazing</b>!</p>"
pattern = r"<[^<]"
matches = re.findall(pattern, text)          # ['<p', '<b', '</', '</']
```

*The above code finds any literal '<' followed by another single character. That single character is defined by the [ ] class, that is, anything that is '<', although, we negate this using the ^, so overall, we say; find two characters where it starts with '<', followed directly by something that ISN'T '<'*
**<p**>This product is **<b**>amazing**</**b>!**</**p>

*The ? makes the + non-greedy (lazy), meaning it will match as few character as possible*

- *Greedy = <.*> = <tag>content</tag>*
- *Non-Greedy = <.*?> = <tag>, </tag>*

**Example**

```
text = "<abc>def<ghi>"
print(e.findall(r"<.*>", text))      # [<abc>def<ghi>]
print(e.findall(r"<.*?>", text))     # [<abc>, <ghi>]
```

*Hence, the greedy pattern does NOT stop until it finds the last instance of '>', while non-greedy will stop once it finds one '>'*

```
# STEP 3: Match any preceding character as long as it's not another '<'
text = "<p>This product is <b>amazing</b>!</p>"
pattern = r"<[^<]+?>"
matches = re.findall(pattern, text)          # ['<p>', '<b>', '</b>', '</p>']
```

```
# STEP 4: To remove all <HTML> tags, we can use the re.sub to replace them with nothing
text = "<p>This product is <b>amazing</b>!</p>"
pattern = r"<[^<]+?>"
matches = re.sub(pattern, "", text)          # This product is amazing!
```

## Validating Emails

```
import pandas as pd
data = {
        'Name' : ['Alice', 'Bob', 'Charlie'],
        'Email' : ['alice@example.com', 'bob.example', 'charlie@domain.com']
}
df = pd.DataFrame(data)
```

```
pattern = r"[^@]+@[^@]+\.[^@]+"
```

```
def validate_email(email):                    # re.match allows us to check the pattern
  if re.match(pattern, email):
    return True
  else:
    return False

df['isValid'] = df['Email'].apply(validate_email)    # Apply allows us to apply function to every entry
```

## RegEx in a Data Frame

```
# Find in the data where the Complaint Type contains the word 'Noise', ignore NA's
noise_complaints = complaints[complaints['Complaint Type'].str.contains('Noise', na=False)]
noise_complaints[['Complaint Type']].head()

# Find complaints that have multiple words in them
pattern = r"Loud|Music"
pattern  = complaints[complaints['Descriptor].str.contains(pattern , na=Flase, regex=True)]
pattern[['Descriptor']]
```

# Data Cleaning

```
# Check for which column has the most missing values
complaints.isnull.sum()
complaints.isnull.sum().sort_values(ascending=False)

# Change types and replace blanks, messy or blank values with NaN
complaints['Closed Date'] = pd.to_datetime(complaints['Closed Date'], errors='coerce')
complaints['Incident Zip'] = pd.to_datetime(complaints['Incident Zip'], errors='coerce')

# Standardise Columns
complaints['Complaint Type'] = complaints['Complaint Type'].str.lower()  # Convert all to lowercase
complaints['Complaint Type'] = complaints['Complaint Type'].str.strip()    # Remove all whitespaces

# Replace missing values of a single column with 'Unknown'
complaints['Descriptor'].fillna('Unknown', inplace=True)
complaints['Incident Zip'].fillna(complaints['Incident Zip'].mean(), inplace=True)

# Replace missing values in multiple columns with 'Unknown'
columns_to_fill = ['Location Type', 'Incident Zip', 'City']
for col in columns_to_fill:
        complaints[col].fillna('Unknown', inplace=True)

# Remove missing values in a single column
complaints.dropna(subset=['Closed Date'])
# Remove missing values where two values in two columns are missing
```

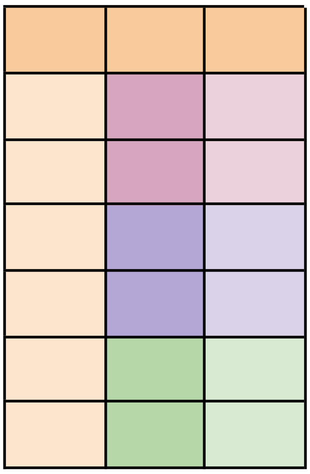
```
no_closed_or_res = complaints.dropna(subset=['Closed Date', 'Resolution Action Updated Date'])

# Drop rows under conditions
cleaned_data = cleaned_data.drop(cleaned_data[cleaned_data['tolls_amount'] <= 0].index)

# Check duplicates and remove duplicate entries
complaints['City'].duplicated().sum()
drop_city_duplicates = complaints.drop_duplicates(subset=['City'])

# Drop rows where a specific column has NULL's or duplicates
complaints.dropna(subset=['City'], inplace=True)

# Removing zips that are outside of a particular range
valid_range_min = 10000
valid_range_max = 10500
complaints['Incident Zip'] = pd.to_numeric(complaints['Incident Zip'], errors='coerce')
complaints = complaints[complaints['Incident Zip'].between(valid_range_min, valid_range_max)]
```

# Grouping, Merging, Reshaping

## Melting

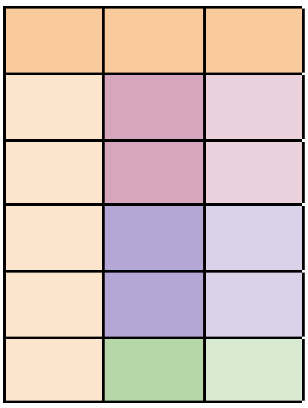
When data sits in a wide format, and putting into a long format (Opposite of pivoting)

→

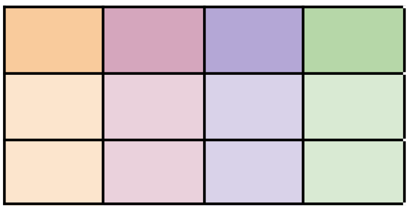


Gather columns into rows

pd.melt(df)



→



Spread rows into columns

df.pivot(columns='var', values='val')

```
import pandas as pd
df = pd.DataFrame({
        'Product': ['A', 'B'],
        'Jan_Sales': [100, 80],
        'Feb_Sales': [150, 120],
        'Mar_Sales': [200, 160]
})
```

| Product | Jan_Sales | Feb_Sales | Mar_Sales |
|---|---|---|---|
| A | 100 | 150 | 200 |
| B | 80 | 120 | 160 |

```
# Melt the dataframe
df = pd.melt(df, id_vars=['Product'], var_name='Month', value_name='Sales')
```

df: your original DataFrame.
id_vars=['Product']: columns to keep fixed (not unpivoted). Here, 'Product' stays as is.
var_name='Month': name for the new column that will hold the original column headers being unpivoted. So the former column names (like months) become values under 'Month'.

| Product | Month | Sales |
|---|---|---|
| A | Jan | 100 |
| A | Feb | 150 |
| A | Mar | 200 |
| B | Jan | 80 |
| B | Feb | 120 |
| B | Mar | 160 |

```
# Pivot the dataframe
df = df.pivot(index='Product', columns='Month', values='Sales').reset_index()
```

| Product | Jan_Sales | Feb_Sales | Mar_Sales |
|---|---|---|---|
| A | 100 | 150 | 200 |
| B | 80 | 120 | 160 |

index='Product': the column to use as the new row labels (rows will be grouped by each unique product).
columns='Month': the column whose unique values will become the new column headers.
values='Sales': the column containing the values to fill the table cells.
Calling .reset_index() after pivots turns the index (Product) back into a regular column.

```
# Melting with the Complaints dataset
melted_complaints = complaints.melt(
        id_vars=['Unique Keys'],
        value_vars=['Complaint Type', 'Descriptor'],
        var_name = 'Category',
        value_name = 'Value')

melted_complaints.head()
```

## Cleaning column names

```
complaints['City'] = complaints['City'].str.upper()
complaints['City'] = complaints['City'].str.strip()
```

## Pivot

```
pivot_table = complaints.pivot_table(
        index='Agency',
        columns='City',
        values='Unique Key',
        aggfunc='count',
        fill_value=0)

pivot_table.sort_values(by='BROOKLYN', ascending=False)
```

## Transpose

```
pivot_table.transpose()

# How can we show the different complaint types occurring in each city?
grouped_complaints = complaints.groupby('City')['Complaint Type'].count()
grouped_complaints.sort_values(ascending=False).head()
```

## GroupBy

```
df = pd.DataFrame({
        'Customer': ['John', 'John', 'Jane', 'Jane'],
        'Product': ['A', 'B', 'A', 'B'],
        'Sales': [100, 150, 200, 250]
})
```

| Customer | Product | Sales |
|---|---|---|
| John | A | 100 |
| John | B | 150 |
| Jane | A | 200 |
| Jane | B | 250 |

df.groupby('Customer').agg({'Sales': 'sum'})

| Customer | Product |
|---|---|
| John | 250 |
| Jane | 450 |

```
# Show different complaint types occurring in each city
grouped_complaints = complaints.groupby('City')['Complaint Type'].count()
grouped_complaints = sort_values(ascending=False).head()

# Count the number of different complaints occurring in each city
grouped_multiple = complaints.groupby(['Complaint Type', 'City'])['Unique Key'].count()
grouped_multiple .sort_values(ascending=False)
```

## Count() vs Size()

*Size includes NaN values, but count does not*

```
complaints.groupby(['City'])['Closed Date'].size().sort_values(ascending=False)
complaints.groupby(['City'])['Closed Date'].count().sort_values(ascending=False)
```

## Merge

Inner join: Everything that overlaps between Group A and Group B
Outer join: Absolutely everything in both Group A and Group B
Left join: Everything in Group A and its overlap with Group B
Right join: Everything in Group B and its overlap with Group A

```
data = {
        'Agency Name': [
                'New York City Police Department',
                'Department of Transportation',
                'Department of Parks and Recreation'
                ],
        'Severity': ['High', 'Medium', 'Low']
}
severity = pd.DataFrame(data)
```

| | Agency Name | Severity |
|---|---|---|
| 0 | New York City Police Department | High |
| 1 | Department of Transportation | Medium |
| 2 | Department of Parks and Recreation | Low |

```
subset = complaints.head(25)
subset['Agency Name'].unique()
```

```
array([
        'New York City Police Department',
        'Department of Health and Mental Hygiene',
        'Department of Transportation'
        ])
```

*We see that 'Department of Health and Mental Hygiene' exists in the subset, but not in the data*

## Inner Join

```
inner_merge = pd.merge(subset, severity, on='Agency Name', how='inner')
print("Rows of data:", len(inner_merge))
inner_merge
```

## Left Join

```
left_merge = pd.merge(subset, severity, left_on='Agency Name', right_on='Agency Name', how='left')
print("Rows of data:", len(left_merge))
left_merge
```

## Right Join

```
right_merge = pd.merge(subset, severity, on='Agency Name', how='right')
print("Rows of data:", len(right_merge))
right_merge
```

## Outter (Full) Join

```
outer_merge = pd.merge(subset, severity, on='Agency Name', how='outer')
print("Rows of data:", len(outer_merge))
outer_merge
```

## Concatenation

*For two different data structures with similar layouts, we should concatenate them together. Instead of merging them, we should concatenate them, adding their rows together.*

```
df1 = complaints.head(5)
```

```
df2 = complaints.tail(5)
concatenated_df = pd.concat([df1, df2], axis=0)
print(len(concatenated_df))
```

## Other

Number of rows of DF: len(df)